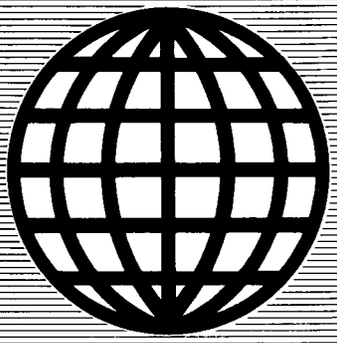


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 39 July / August 1989 $3.00

Programming for Performance
Assembly Language Techniques

Computer Aided Publishing
The Hewlett Packard LaserJet

The Z-System Corner
System Enhancements with NZCOM

Generating LaserJet Fonts
A Review of Digi-Fonts

Advanced CP/M
Making Old Programs Z-System Aware

C Pointers, Arrays & Structures Made Easier
Part 3: Structures

Shells
Using ARUNZ alias with ZCAL

Real Computing
The National Semiconductor NS320XX

ISSN # 0748-9331

THE COMPUTER JOURNAL

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Bruce Morgan
Richard Rodman
Jay Sage

The Computer Journal is published six times a year by Publishing Consultants, 190 Sullivan Crossroad, Columbia Falls, MT 59912 (406) 257-9119

Entire contents copyright© 1989 by Publishing Consultants.

***Subscription rates**—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in US dollars on a US bank.*

Send subscriptions, renewals, or address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912.

Address all editorial and advertising inquiries to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912 phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 39 July / August 1989

Editorial **2**

Programming for Performance **3**
Using special assembly language techniques to write space saving high performance Z80 code. By Lee A. Hart.

Computer Aided Publishing **9**
First in a series on PCL programming and soft font handling for the Hewlett Packard LaserJet. By Art Carlson.

The Z-System Corner **12**
System enhancements using NZCOM. By Jay Sage.

Generating LaserJet Fonts **15**
A review of the Digi-Fonts system for generating custom soft fonts for the LaserJet. By Art Carlson.

Advanced CP/M **19**
Adding Z-System capability to programs written in C or Pascal and making old programs Z-System aware. Z80 interrupt bug and detecting interrupt status. By Bridger Mitchell.

C Pointers, Arrays and Structures Made Easier **23**
Part 3: Structures. By Clem Pepper.

Shells **28**
Using an ARUNZ alias with ZCAL. By Rick Charnes.

Real Computing **32**
The National Semiconductor NS329XX. By Richard Rodman.

Computer Corner **40**
By Bill Kibler.

News **34**

Editor's Page

DTP Trauma

We had planned on gradually converting to Desk Top Publishing, in fact the Editorial and Computer Aided Publishing article in issue #38 were produced with DTP. Our intentions were to use both the old phototypesetter and the new DTP systems in parallel, doing a few more articles on each issue with DTP while we became comfortable with DTP. There was no rush, because I figured that it would take six to eight months to sell the 10 year old typesetter.

The material for this issue was all prepared and coded for the typesetter when I placed the first ad to try to sell the old equipment--I wasn't worried, after all it would take a long time. Four days later, a buyer showed up! We made a deal, and he took the typesetter. When someone offers cash for a piece of equipment which is becoming obsolescent you don't tell him to come back in three or four weeks, because he might change his mind and it could be a very long time before another buyer appeared. So there we were. It was the week to set type and paste up, but everything was coded for the old typesetter which was no longer here.

The only answer was to strip out all of the code and change over to PageMaker and the LaserJet. Setting it up under PageMaker is actually much easier than the old system for about 95% of the material. I'm currently using WordStar 4.0 which does not have style sheets which will transfer over to PageMaker, so I set up PageMaker templates and embedded style tags in the WordStar files.

It was a crash learning course which took quite a few hours and delayed getting this issue to the printer--but it was well worth it. It will take much less time to prepare future issues, and we may even be able to send galleys to the authors for proofreading.

There are still a few rough spots which I have not had time to take care of. For example, I couldn't get an EM dash to transfer from WordStar to PageMaker for the HP LaserJet. There were also problems in retaining the indent levels for the code sections.

I am in the process of changing over to MicroSoft Word Version 5.0 (more learning time). I'll generate fonts with custom symbol sets using Digi-Fonts, then install the same fonts in both Word and PageMaker. This should solve the problems in showing the same extended characters in both Word and PageMaker. The industry needs to standardize so that all applications can refer to one soft font file so that the fonts do not have to be installed in every one of the applications--it is a nuisance and consumes valuable disk space.

Material which is Desk Top Published should not look significantly different that something which is published by traditional methods. Hopefully the only difference you'll notice is that we are using slightly different type faces for this issue. We'll be interested in your comments after we gain experience and produce three or four issues on DTP. One change you will notice is the vast improvement in hyphenation in PageMaker compared with that which we used to get from the Compugraphic phototypesetter.

I am comfortable with the combination of MicroSoft Word, PageMaker, Digi-Fonts, and the LaserJet for producing the magazine. With PageMaker it is a real joy to push columns of text around while making things fit. It is so much faster than the mechanical pasteup we have been doing. Once you've tried it, there is no going back.

But, DTP is not the complete answer to everything. I still intend to write programs to handle some of the unusual requirements. One thing that I'd like is a utility which would analyze a PM3 file and print out the parameters such as margins, columns, style definitions, etc. I like to keep a hard copy record what I've done.

Another needed utility is one which would output pages with the correct imposition (the arrangement of pages in the proper order for printing a signature) for binding. When printing a 40 page saddle stitched 5.5 by 8.5 inch book two up on 8.5 by 11 inch paper, pages 1 and 40 are printed on one sheet, and pages 2 and 39 on the other side. the next sheet has pages 3 and 38 on one side and 4 and 37 on the other side. After folding and binding, the pages read in correct order. I'd like to use PageMaker for the page preparation, and then use the utility to output the pages in the desired order.

I need to analyze the PageMaker PM3 file structure to determine how to extract the information for these utilities. Does anyone have any information on the PM3 file structure?

I'll also be working on runoff programs which talk directly to the LaserJet for applications where PageMaker's WYSIWYG screen display is not required (such as setting a book from a CP/M system using an ASCII character terminal). Another area of activity will be laser typesetting from database files using dBXL, and C.

Send in your questions, problems, tips and solutions.

Users' Toolkits

Successful systems in the near future are going to have to be customizable to do what the user wants, the way the user wants to do it. As stated on page one of volume 1 issue 11, of *UNIX Journal* (7620 242nd Street S.W., Edmonds, WA 98020-5463) "Products of the future are going to [be] tailored to the needs of individual

(Continued on page 37)

Programming For Performance

by Lee A. Hart

Over the years, the ancient masters of the software arts have meticulously crafted the tools of structured programming. They have eloquently preached the virtues to body and soul that come from writing clean, healthy code, free from the evils of self-modifying code or the dreaded GOTO.

Many programmers have seen the light. They write exclusively in structured high-level languages, and avoid BASIC as if it carried AIDS. Assembly language is just that unreadable stuff the compiler generates as an intermediate step before linking. Memory and processor speed are viewed as infinite resources. Who cares if it takes 100K for a pop-up calculator program? And if it's not fast enough, use turbo mode, or a 386.

But a REAL pocket calculator doesn't have a 16-bit processor, or 100K of RAM; it typically runs a primitive 4-bit CPU at 1 MHz or less, with perhaps 2K of memory. Yet it can out-perform a PC clone having 10 times the speed and memory!

How can this be? Special hardware? Tricky instruction sets? On the contrary; CPU registers and instructions have instead been removed to cut cost. No; the surprising performance comes from clever, efficient programming with an extreme attention to detail. Such techniques are essential to the success of every high-volume micro-based product. But they aren't widely known and so are rarely applied to general-purpose microcomputers.

Suppose your micro doesn't provide the luxury of unlimited (or even adequate) resources. Your program absolutely has to fit in a certain space, such as a ROM. You're stuck with a slow CPU but must handle a hardware device with particularly severe timing requirements. Your C compiler just turned out a program that misses the mark by a megabyte. Don't give up! I'll show you some techniques that are particularly effective at "running light without overbyte," as Dr. Dobbs used to say.

I'll demonstrate these techniques with the Z80. With over 30 million sold last year alone, it remains the number-one-selling micro and is widely used in cost-effective designs when performance counts. However, the principles used apply to almost any microcomputer.

In the beginning

Novice Z80 programmers soon spot peculiarities in the instruction set; arcane rules restrict data movement between registers. For instance, the stack pointer can be loaded but not examined. Flags are not set automatically and must be explicitly updated by a math or logical instruction. The carry flag can be set or inverted but not reset. Of the six flags, only four can be tested by jumps and calls (and only two by relative jumps).

These limitations are no accident. They represent an artful compromise between cost, complexity, performance, and compatibility with the earlier 8080 instruction set. To get the most out of any micro, you must discover how its designer expected you to use the architecture. Get "inside" his head; become part of the machine.

The Z80 is register-oriented; it manipulates data in registers efficiently but deals rather clumsily with memory. Registers are specialized, with each having an intended purpose. Here are some rules I've found useful:

A = Accumulator: first choice for 8-bit data; best selection of load/store instructions; source and destination for most math, logical, and comparisons.

HL = High/Low address: first choice for 16-bit data/addresses; source and destination for 16-bit math; second choice for 8-bit data; pointer when one math/logical/compare operand is in memory; source address for block operations.

DE = DEstination: second choice for 16-bit data/addresses; third choice for 8-bit data; destination address for block operations.

BC = Byte Counter: third choice for 16-bit data/addresses; I/O port addreses; 8/16 bit counter for loops and block operations.

F = Flag byte (6 bits used): updated by math/logical/compare instructions; Zero, Carry, Sign, and Parity tested by conditional jumps, calls, or returns; Zero and Carry by relative jumps; block operations use Parity; bit tests use Zero; shifts use Carry; only decimal adjust tests Half-carry and Add/Subtract flags.

A',BC',DE',F',HL' = twins of A, BC, DE, F, HL; can be quickly swapped with main set; use for frequently used variables, fast interrupt handlers, task switching.

R = Refresh counter for dynamic RAM: also counts instructions for diagnostics, debuggers, copy-protection schemes; pseudorandom number generator; interrupt detection.

I = Interrupt vector: page address for interrupts in mode 2; otherwise, an extra 8-bit register that updates flags when read.

IX,IY = Index registers X and Y: Two 16-bit registers, used like HL as an indirect memory pointer, except instructions can include a relative offset.

SP = Stack Pointer: 16-bit memory pointer for LIFO (last-in first-out) stack to hold interrupt and subroutine return address, pushed/popped register data; stack-oriented data structures.

Naturally, some instructions get used a lot more than others. But frequency-of-use studies reveal that many programs NEVER use large portions of the instruction set. Sometimes there are good rea-

sons, like sticking to 8080 opcodes so your code runs on an 8080/8085/V20 etc. More often the programmer is simply unfamiliar with the entire instruction set, and so restricts himself to what he knows.

This is fine for noncritical uses but suicidal when performance counts. It's like running a racehorse with a gag in its throat. Take some time to go over the entire instruction set, one by one. Devise an example for each instruction that puts it to good use. I only know of nine turkeys with no use besides "trick" NOPs (can you find them?).

Figure 1 shows a routine that might be written by a rather inept programmer (or an unusually efficient compiler). It outputs a string of characters ending in 0 to the console. It generally follows good programming practices; it's well structured, has clearly-defined entry and exit points, and carefully saves and restores all registers used.

Now let's see how it can be improved. First, note that over half the instructions are PUSHes or POPs. This is the consequence of saving every register before use, a common compiler strategy. Though safe and simple, it's the single worst performance-killer I know.

The alternative is to push/pop only as necessary. This is easier said than done; miss one, and you've got a nasty bug to find. A good strategy helps. I initially define my routines to minimize the registers used; only push/pop as needed within the routine itself; and restore nothing on exit. In OUTSTR, this eliminates all but the PUSH DE/POP DE around the CALL BDOS.

This shifts the save/restore burden to the calling routine. Since the caller also follows the rule of minimal register usage and push/pops only as necessary, it will probably not push/pop as many registers; thus we have increased speed by eliminating redundant push/pops. We have also made it explicitly clear which registers a caller really needs preserved.

Now I move the remaining push/pops to the called routines to save memory. If every caller saves a particular register, it obviously should be saved/restored by the subroutine itself. If two or more callers save it, speed is the deciding factor; preserve that register in the subroutine if the extra overhead is not a problem for callers that don't need that register preserved.

Push/pops are slooooww; at 21 to 29 T-states per pair, they make wonderful low-byte time killers. If possible, either use, or save to, a register that isn't killed by the called routine. In our example, try IX or IY instead of DE; the index registers aren't trashed by the BDOS call (except, see Jay Sage's column. Ed). This saves 5 T-states/loop but adds 2 bytes (see why?). The instruction EX DE,HL (8 T-states per pair) is often useful, but not here; the BIOS eats both HL and DE. The ultimate speed demon is a fast-n-drastic pair of EXX instructions to replace the PUSH DE/POP DE. They save 13 T-states with no size increase, and even preserve BC so we don't have to reload it for every loop.

```
        ...
        ld      de,string       ; point to message
        call    outstr          ; output it
        ...

outstr: push    af              ; save registers
        push    bc
        push    de
        push    hl
        ld      a,(de)          ; get next character
        cp      0               ; compare it to 0
        jp      z,outend        ; if not last char,
        push    de              ; ..save registers
        ld      e,a
        ld      c,conout        ; output character to console
        call    bdos
        pop     de              ; restore registers
        inc     de              ; advance to next
        jp      outstr          ; repeat until done
outend: pop     hl              ; else 0 is last char
        pop     de
        pop     bc              ; restore registers
        pop     af
        ret                     ; return

string: db      'message'       ; display our message
        db      0               ; end of string marker
```

Figure 1: A routine to output a srting of chars.

```
    and  a      ; update flags and clear Carry
    xor  a      ; set A=0, update flags, P/V flag=1
    sub  a      ; same, but P/V flag=0
    sbc  a,a    ; set all bits in A to Carry (00 or FF)
    add  a,a    ; A*2, or shift A left & set lsb=0
    add  hl,hl  ; HL*2, or shift HL left & set lsb=0
    adc  hl,hl  ; shift HL left & lsb=Carry
    sbc  hl,hl  ; set all bits in HL to Carry (0000 or FFFF)
    ld   hl,0   ; \_load SP into HL so it can be examined
    add  hl,sp  ; /
```

Figure 2: Side effects of some not-so-obvious instructions.

Comparisons

A CP 0 instruction was used to test for 0, an obvious choice. But it takes 2 bytes and 7 T-states to execute. The Z80's Zero flag makes the special case of testing for zero easy; all we have to do is update the flags to match the byte loaded. This is most easily done with an OR A instruction, which takes only 1 byte and 4 T-states. You'll find this trick often in Z80 code.

Note that OR A has no effect on A; we just used it to update the flags because it's smaller and faster than CP 0. This illustrates a basic principle of assembly languages; the side effects of an instruction are often more important than the main effect. Some other not-so-obvious instructions are shown in Figure 2.

Using DE as the string pointer is a weak choice. It forces us to load the character into A, then move it to E. If we use HL, IX, or IY instead, we can load E directly and save a byte. But this makes it harder to test for 0.

An INC E, DEC E updates the Z flag without changing E. Or mark the end of the string with 80h, and use BIT 7,E to test for end. Both are as efficient as the OR A trick but don't need A. If you are REALLY desperate, add 1 to every byte in the string, so a single DEC E restores the character and sets the Z flag; kinky, but short and fast.

Jumps

This example used 3-byte absolute jump instructions. We can save memory by using the Z80's 2-byte relative jumps instead; each use saves a byte. Since jumps are among the most common instructions, this adds up fast.

Relative jumps have a limited range, so it pays to arrange your code carefully to maximize their use. I've found that about half the jumps in a well structured program can be relative. When most of the jumps are out of range, it's often a sign of structural weaknesses, "spaghetti-code" or excessively complex subroutines.

How about execution speed? An absolute jump always takes 10 T-states; a relative jump takes 12 to jump, or 7 to continue. So if speed counts, use absolute jumps when the branch is normally taken, and relative jumps when it is not. In the example, this means changing the JP Z,OUTEND to JR Z,OUTEND but keeping the JP at the end.

But wait a minute! The JR Z,OUTEND merely jumps to the RET at the end of the subroutine. It would be more efficient still to replace it with RET Z, a 1-byte conditional return that is only 5 T-states if the return is not taken. This illustrates another difference between assembler and high-level languages; entry and exit points are often not at the beginning and end of a routine.

We can speed up unconditional jumps within a loop. On entry, load HL with the start address of the loop, and replace JP LABEL by JP (HL). It takes 1 byte and 4 T-states, saving 6 T-states per loop. This scheme costs us a byte (+3 to set HL; -2 for JP (HL)). But if used more than once in the routine, we save 2 bytes per occurrence. If HL is unavailable (as is the case here; the BDOS trashes it), IX or IY can be used instead. However, the JP (IX) and JP (IY) instructions take 2 bytes and 8 T-states, making the savings marginal.

Can we do better yet? Yes, if we carefully rethink the structure of our program. Notice it has two jump instructions per loop; yet only one test is performed (test for 0). This is a hint that one conditional jump should be all we need. Think of the instructions in the loop as links in a chain. Rotate the chain to put the test-for-0 link at the bottom, and LD C,CONOUT on top (which we'll label OUTNXT). The JP OUTSTR is now unnecessary, and can be removed. JP NZ,OUTNXT performs the test and loops until 0 (remember, absolute for speed, relative for size). The entry point is still OUTSTR, though (horrors!) it's now in the middle of the routine.

```
        ...
        ld      hl,string           ; point to message
        call    outstr              ; output it
        ...

outstr: ld      b,(hl)              ; get length of message
        ld      c,conout            ; output to console
        ld      e,(hl)              ; get 1st char
outnxt: exx                         ; save registers,
        call    bdos                ;   output char,
        exx                         ;   and restore
        inc     hl                  ; advance to next
        ld      e,(hl)              ; get next character
        djnz    z,outnxt            ; loop until end
        ret

string: db      strend - strbeg     ; message length
strbeg: db      'message'           ; message itself
strend:
```

Figure 3: Using INC L instead of INC HL to save 2 T-states.

```
        ...
        call    outstr              ; output message
        dw      string              ; beginning here
        ...

outstr: pop     hl                  ; get pointer to "DW STRING"
        ld      e,(hl)              ;   E=low byte of string addr
        inc     hl
        ld      d,(hl)              ;   D=high byte of string addr
        inc     hl                  ; skip over "DW STRING" & push
        push    hl                  ;   corrected return address

outnxt: ld      a,(de)              ; get next character
        or      a                   ; if 0,
        ret     z                   ;   all done, return
        push    de
        ld      e,a
        ld      c,conout            ; output character to console
        call    bdos
        pop     de
        inc     de                  ; advance to next
        jr      outnxt
```

Figure 4: Passing parameters as "data" bytes.

We've also made a subtle change in the logic. Presumably we wouldn't call OUTSTR unless there was at least one character to output. But what would happen if we did?

Another way is to use DJNZ to close the loop. Make the first byte of the string its length (1-256). Load this value into B as part of the initialization. The resulting program takes 34 T-states per loop (not counting the CALL).

STILL faster? OK, you twisted my arm. If you're absolutely sure the string won't cross a page boundary, you can use INC L instead of INC HL to save 2 T-states. The 8-bit INC/DEC instructions are faster than their 16-bit counterparts, but should only be used if you're positive the address will never require a carry. This brings us to 32 T-states/loop (see Figure 3), which is the best I can do within this routine itself. Or can you do better?

Parameter Passing

In the above example, parameters were passed to the subroutine via registers (string address in HL). This is fast and easy, but each call to OUTSTR takes 6 bytes. Now let's look at methods that save memory at the expense of speed.

Parameters can be passed to a subroutine as "data" bytes immediately following the CALL. Let's define the two bytes after CALL OUTSTR as the address of the string. The code shown in Figure 4 then picks up this pointer, saving us a byte per call. The penalty is in making OUTSTR 4 bytes longer and 38 T-states/loop slower; thus it doesn't pay until we use it 5 or more times.

We also had to rethink our choice of registers. If we tried to use HL or IX as the string pointer, OUTSTR would have been larger and slower (try it yourself). This demonstrates the consequences of inappropriate register choices.

The more parameters that must be passed, the more efficient this technique becomes. A further refinement is to put the string itself immediately after CALL as shown in Figure 5. This saves an additional two bytes per call, and shortens OUTSTR by 6 bytes.

Constants and Variables

Constants and variables are part of every program. Constants are usually embedded within the program itself, as "immediate" bytes. Variables on the other hand are usually separated, grouped into a common region perhaps at the end of the program. This makes sense for programs in ROM, where the variables obviously must be stored elsewhere. But it is not a requirement for programs in RAM.

If your program executes from RAM, performance can be improved by treating variables as in-line constants; storage for the variable is in the last byte (or two) of an immediate instruction. The example in Figure 6 is a routine that creates a new stack, toggles a variable FLAG between two states, and then restores the original stack.

The LD A,(FLAG) instruction takes 13 T-states and 4 bytes of RAM (3 for the instruction, 1 to store FLAG). It can be replaced by LD A,'Y' where 'Y' is the initial value of the variable FLAG, the 2nd byte of the instruction (see Figure 7). Speed and memory are improved 2:1, to 7 T-states and 2 bytes respectively.

It works for 16-bit variables as well. Replace LD SP,(STACK) with LD SP,0 where 0 is a placeholder for the 2-byte variable STACK. This saves 3 bytes and 10 T-states.

There is another advantage to this technique--versatility. Any immediate-mode instruction can have variable data; loads, math, compares, logical, even jumps and calls. Try changing our first example so a variable OUTDEV selects the output device; console or printer. Now see how simple it is if OUTDEV is the 2nd byte of the LD C,CONOUT instruction.

It even creates new instructions. For instance, the Z80's indexed instructions don't allow a variable offset. This makes it awkward to load the "n"th byte of a table, where we would like LD A,(IX+b) where "b" is a variable. But it can be done if the variable offset is stored in the last byte of the indexed instruction itself.

Storing variables in the address field of a jump or call instruction can do some weird and wonderful things. There is no faster way to perform a conditional branch based on a variable. But remember you are treading on the thin ice of self-modifying code; debugging and relocation become much more difficult, and you must insure that the variable NEVER has an unexpected value. Also, in microprocessors with instruction caches (fast memory containing copies of the contents of regular memory), there can be problems if the cache data are not updated.

```
        ...
        call    outstr          ; output message
        db      'message',0     ; which immediately follows
        ...

outstr: pop     de              ; get pointer to message
        ld      a,(de)          ; get next character
        inc     de              ; advance to next
        push    de              ; & save as return address
        or      a               ; if char=0, all done
        ret     z               ;   pointer is return addr
        ld      e,a
        ld      c,conout        ; else output char to console
        call    bdos
        jr      outstr          ;   & repeat
```

Figure 5: Placing the string immediately after CALL.

```
toggle: ld      (stack),sp      ; save old stack pointer
        ld      sp,mystack      ; setup my stack
        ld      a,(flag)        ; get Yes/No flag
        cp      'Y'             ; if "Y",
        ld      a,'N'           ;    set it to "N"
        jr      z,setno         ; else "N",
        ld      a,'Y'           ;    set it to "Y"
setno:  ld      (flag),a        ; save new state
        ld      sp,(stack)      ; restore stack pointer
        ret

stack:  dw      0               ; old stack pointer
flag:   db      'Y'             ; value of flag
```

Figure 6: A routine which creates a new stack, toggles a variable FLAG, then restores the stack.

```
toggle: ld      (stack+1),sp    ; save old stack pointer
        ld      sp,mystack      ; setup my stack
flag:   ld      a,'Y'           ; get Y/N flag (byte 2=var)
        cp      'Y'             ; if "Y",
        ld      a,'N'           ;    set it to "N"
        jr      z,setno         ; else "N",
        ld      a,'Y'           ;    set it to "Y"
setno:  ld      (flag+1),a      ; save new state
stack:  ld      sp,0            ; restore stack (byte 2,3=var)
        ret
```

Figure 7: Treating variables as in-line constants for programs which execute in RAM.

```
toggle: ld      (stack+1),sp    ; save old stack pointer
        ld      sp,mystack      ; setup my stack
flag:   ld      a,'Y'           ; get Y/N flag (byte 2=var)
        xor     'Y'-'N'         ; toggle "Y" <-> "N"
        ld      (flag+1),a      ; save new state
stack:  ld      sp,0            ; restore stack (byte 2,3=var)
        ret
```

Figure 8: Using XOR to toggle a variable.

I put a LABEL at each instruction with an immediate variable, then use LABEL+1 for all references to it. This serves as a reminder that something odd is going on. Be sure to document what you're doing, or you'll drive some poor soul (probably yourself) batty.

Exclusive OR

The XOR operator is a powerful tool for manipulating data. Since anything XOR'd with itself is 0, use XOR A instead of LD A,0. To toggle a variable be-

tween two values, XOR it with the difference between the two values. Our last example can be performed much more efficiently by using XOR as shown in Figure 8.

XOR eliminated the jump, for a 2:1 improvement in size and speed. This illustrates a generally useful rule. Almost any permutation can be performed faster, without jumps, by XOR and the other math and logical operators. Consider the routine shown in Figure 9 to convert the ASCII character in A to uppercase: it's both shorter and faster than the traditional method using jumps.

Data Compaction

Programs frequently include large blocks of text, data tables, and other non-program data. Careful organization of such information can produce large savings in memory and speed of execution.

ASCII is a 7-bit code. The 8th bit of each byte is either unused or just marks the end of a string. You can bit-pack 8 characters into 7 bytes with a suitable routine. If upper case alone is sufficient, 6 bits are enough. For dedicated applications, don't overlook older but more memory-efficient codes like Baudot (5 bits), EBCD (4 bits), or even International Morse (2-10 bits, with frequent characters the shortest).

If your text is destined for a CRT or printer, it may be heavy on control characters and ESC sequences. I've found the following algorithm useful. Bytes whose msb=0 are normal ASCII characters: output as-is. Printable characters whose msb=1 are preceeded by ESC, so "A"+80h=C1h sends "ESC A". Control codes whose msb=1 are a "repeat" prefix to output the next byte between 2 and 32 times. For example, linefeed+80h=8Ah repeats the next character 11 times. The value 80h, which otherwise would be "repeat once", is reserved as the marker for the end of the string.

Programs can be compacted, too. One technique is to write your program in an intermediate language (IL) better suited to the task at hand. It might be a high-level language, the instruction set of another CPU, or a unique creation specifically for the job at hand. The rest of your program is then an interpreter to execute this language. Tom Pittman's Tiny BASIC is an excellent example of this technique. His intermediate language implemented BASIC in just 384 bytes; the IL interpreter in turn took about 2K.

Another approach is threaded code,

```
convert: ld    b,a             ; save a copy of the char in B
         sub   'a'             ; if char is lowercase (a thru z),
         cp    'z'-'a'+1       ;   then carry=1, else carry=0
         sbc   a,a             ;   fill A with carry
         and   'a'-'A'         ;   difference between upper/lower
         xor   b               ;   convert to uppercase
```

Figure 9: Converting ASCII char to uppercase with XOR.

```
main:   call  getname
        call  openfile
        call  readfile
        call  expandtabs
        call  writefile
        call  closefile
        ret
```

Figure 10: Example of threaded code which uses lots of CALLs.

```
main:   ld    (stack),sp ; save stack pointer
        ld    sp,first   ; point it to first address in the list
        ret              ; and go execute it
first:  dw    openfile
        dw    readfile
        dw    expandtabs
        dw    writefile
        dw    closefile
        dw    return     ; end of list

return: ld    sp,(stack) ; restore stack pointer
        ret              ; and return (to MAIN's caller)
```

Figure 11: Eliminating the CALL opcodes.

```
main:   call  next
        dw    openfile
        dw    readfile
        dw    expandtabs
        dw    writefile
        dw    closefile
        dw    return     ; end of list

next:   pop   ix         ; make IX our next-subroutine pointer
next1:  ld    hl,next1   ; push address so RET comes back here
        push  hl
        ld    l,(ix+0)   ; get address to "call"
        inc   ix         ;    low byte
        ld    h,(ix+0)   ;    high byte
        inc   ix         ; point IX to addr for next time
        jp    (hl)       ; call address

return: pop   hl         ; end of list; discard NEXT addr
        ret              ; and return to MAIN's caller
```

Figure 12: Indirectly threaded code linked by NEXT.

made popular by the Forth language. A tight, well-structured program will probably use lots of CALLs. At the highest levels, the code may in fact be nothing but long sequences of CALLs (see Figure 10).

Every 3rd byte is a CALL; large programs will have 1000s of them. So let's eliminate the CALL opcodes, making our program just a list of addresses (see Figure 11.)

The stack pointer is pointed to the address of the first subroutine in the list to execute. RET then loads this address into the program counter and advances the stack pointer to the next address. Since each subroutine also ends with a RET, it automatically jumps directly to the next routine in the list to be executed. This is called directly threaded code.

RETURN is always the last subroutine in a list. It restores the stack pointer and returns to the caller of MAIN.

Directly threaded code can cut program size up to 30%, while actually increasing execution speed. However, it has some rather drastic limitations. During execution of the machine-code subroutines in the address list, the Z80's one and only stack pointer is tied up as an address pointer. That means the stack can't be used; no interrupts, calls, pushes, or pops are allowed without first switching to a local stack.

The solution to this is called indirectly threaded code, made famous (or infamous) by the Forth language. Rather than have each subroutine directly chain into the next, they are linked by a tiny interpreter, called NEXT (see Figure 12).

Now IX is our pointer into the address list; it points to the next subroutine to be executed. Subroutines can use the stack normally within, but must preserve IX and can't pass parameters in HL. When they exit via RET, it returns them to NEXT1.

Though the example executes the address list as straight-line code, subroutines can be written to perform jumps and calls via IX as well. NEXT can provide special handling for commonly-used routines as well; words with the high byte=0 could jump IX by a relative offset if A=0. If there are less than 256 subroutines, each address can be replaced by a single byte, which NEXT converts into an address via a lookup table.

Indirectly threaded code can reduce size up to 2:1 in return for a similar loss in execution speed. The decrease in program size is often remarkable. I learned this in 1975 designing a sound-level dosimeter. This cigarette-pack sized gadget rode around in a shirt pocket all day, logging the noise a person was exposed to. It then did various statistical computations to report the high, low, mean, and RMS noise levels versus time.

In those dark ages, a BIG memory chip was 256x4. Cost, power, and size forced us into an RCA 1802 CMOS microprocessor, with just 512 bytes of program memory (bytes, not K!). Try as we might, we couldn't do it. In desperation, we tried Charlie Moore's Forth. Incredibly, it bettered even our heavily optimized code by 30%!

Of course, very little of Forth itself wound up in the final product; it just showed us the way. Once you know HOW it's done, you can apply the same techniques to any assembly-language program without becoming a born-again Forth zealot.

Shortcuts

Here are some "quickies" that didn't fit in elsewhere. Keep in mind what is actually in all the registers as you program. Do you really need to clear carry, or is it already cleared as the result of a previous operation? Before you load a register, are you sure it's necessary? Perhaps it's already there, or sitting in another register.

```
clear1:   ld   a,1        ; clear 1 byte
          db   21h        ;   skip next two bytes (21h = ld hl,nn)
clear80:  ld   a,80       ; clear 80 bytes (and "nn" for ld hl,nn)
          db   26h        ;   skip next byte (26h = ld h,n)
clear256: xor  a          ; clear 256 bytes (and "n" for ld h,n)

clear:    ld   b,a        ; clear #bytes in A to zero
          ld   hl,buffer  ;   beginning at buffer
loop:     ld   (hl),0
          inc  hl
          djnz loop
          ret
```

Figure 13: Simulating a skip.

Many routines return "leftovers" that can be very useful, such as HL, DE, and BC=0 after a block move. Perhaps an INC or DEC will produce the value you want. Variables can be grouped so you needn't reload the entire address for each. If the high byte of a register is correct, just load the lower half.

Keep an index register pointed to your frequently-used variables. This makes them easier to access (up to 256 bytes) and opens the door to memory-(rather than register-) oriented manipulations. The indexed instructions are slower and less memory-efficient, but the versatility sometimes makes up for it (store an immediate byte to memory, load/save to memory from registers other than A, etc.).

The Z80's bit test/set/reset instructions add considerable versatility if you define your flags as bits rather than bytes. Bit flags can be accessed in any register, or even directly in memory, without loading them into a register.

If the last two instructions of a subroutine are CALL FOO and RET, you could just as well end with JP FOO and let FOO do the return for you. If the entry point of FOO is at the top, even the JP is unnecessary; you can locate FOO immediately after and "fall in" to it.

If you have a large number of jumps to a particular label (like the start of your MAIN program), it may be more efficient to push the address of MAIN onto the stack at the top of the routine. Each JP MAIN can then be replaced by a 1-byte RET.

SKIP instructions are a short, fast way to jump a fixed distance. The Z80 has no skips, but you can simulate a 1- or 2-byte skip with a 2- or 3-byte do-nothing instruction: JR or JP on a condition that is never true, for instance. If the flags aren't in a known state, load to an unused register (see Figure 13).

The stack pointer is the Z80's only auto-increment/decrement register. This makes it uniquely suitable for fast block operations. For instance, the fastest way to clear a block of RAM is to make it the stack and push the desired data. At 11 T-states per 2 bytes, it is 3 times faster than two LDD instructions. Remember to disable interrupts or to allow for them; if an interrupt routine pushes onto the stack while you are using it for this special purpose, the results may not be what you intended.

That is all for this time. Next time we will continue the discussion with a look at the interplay between software and hardware. •

Computer Aided Publishing

The Hewlett Packard LaserJet

by Art Carlson

Computer historians will note three products which had tremendous impact on the microcomputer industry. The first is the Apple II which demonstrated what microcomputers could do. The second is VisiCalc, the first electronic spreadsheet, which created a demand for desktop computers in business. The third is the Hewlett-Packard LaserJet Printer which changed business printing and laid the ground work for DTP (Desk Top Publishing).

Before the laser, hardcopy output was limited to high speed lineprinters which produced poor quality printing suitable for mailing labels, slow daisy wheel printers which produced high quality printing for letters, and the ubiquitous dot matrix printer with different models providing a wide range of speeds and poor to moderate quality. None of these could produce the economical high quality *camera ready copy* which publishers needed--their only recourse was expensive time consuming typesetting services for the straight text type matter. Graphics required skilled pen and ink artists, which was too expensive for most publications.

The Hewlett-Packard LaserJet

The LaserJet, which was introduced in 1984, has revolutionized business and publishing practices, and the transition will accelerate as more people understand how to incorporate laser printers into their operations.

Although DTP is receiving most of the press coverage, it is the hidden business applications which account for the majority of the sales--and also the majority of the programming job opportunities. Incidentally, I don't know who coined the term *Desk Top Publishing*, but it should really be *Desk Top Typesetting*. DTP does offer the solution to a very vexing problem, but it only addresses a portion of the total publishing effort.

The LaserJet II series can print on various sizes including 8.5 x 11, 8.5 x 14, and number 10 business size envelopes. It can use plain paper (including a super-smooth grade for reproduction master copy), transparency, and label stock.

The Hewlett-Packard PCL printers use fonts which are already in bit map form, and there are many type options. The LaserJet II comes with Courier, Courier Bold, and Line_Printer in ROM for both portrait and landscape (portrait is across the narrow dimension and landscape is across the wide dimension). There are two slots for ROM cartridges for additional fonts with a wide selection of cartridges available from H-P and other vendors. *Soft fonts* which can be uploaded into RAM offer an almost infinite variety, limited only by the available memory in the printer.

These fonts are all in bit map form and can only be used exactly as they exist. Any scaling or other modifications must be performed before they are loaded into the printer.

Graphics are also sent in bit map form, and can use a lot of memory. The ROMs include graphic routines for drawing filled rectangles, and the most efficient way to draw horizontal and vertical lines is using graphics rules (solid-filled rectangular areas).

Tasks which must be performed repeatedly can be incorporated in a macro which is sent to the printer once and then initiated with a single command. This can greatly reduce the transmission time for business applications.

The LaserJet can produce high quality text, forms, and graphics. While the hardware exists, there is still a lot of programming effort required in order to fully utilize the LaserJet's capabilities.

Programming the LaserJet

The LaserJet is programmed using H-P's PCL (Printer Control Language). Printing simple text files is easy. In fact you don't have to program at all in order to print letters using the internal ROM fonts or cartridge fonts. All you have to do is to use the control panel to select the desired font and then send the printer a stream of ASCII text. For example, when I want to see a hex/ASCII dump of a file for debugging, I use the control panel to select the line_printer font. I then load the file into DDT or DEBUG and enter a Control P so that the output is directed to the printer. This works on both CP/M and MS-DOS computers. Printing manuscripts and letters is just as easy using various runoff programs and editors which don't even know that they are talking to a laser.

When used this way, the LaserJet is merely acting as a high-speed high-quality printer. It's only using a fraction of its capabilities, but thousands are being used just this way by lawyers, CPAs, and businesses to produce correspondence, forms, and reports. While cranking out boilerplate copy using a single fixed width font is easy, the extra effort required to use proportionally spaced fonts in various sizes is well worth while.

The biggest problem facing programmers and business people is understanding the concept of proportionally spaced type. We are so accustomed to typewriters and computer screens with fixed width characters that it is very difficult to accept the fact that every character in a font can be a different width. And that the same character in a different point size is a different width. And even that the same character in the same point size may have different widths in different faces. And that the space has an undefined width which can vary between certain limits in order to justify a line (that's why the space

is never used for indenting or column positioning with proportional fonts). I spend a lot of time explaining and demonstrating proportional spacing, and then the people still count characters and use multiple spaces--it's very difficult to change such ingrained habits.

Working with proportional fonts is not difficult, it's just different. The first step is to forget about talking about line lengths in characters. Think about them in inches or millimeters or picas, or any measure of length. Next convert the line length to units for the laser printer. Then determine the set width for the character you are using and subtract that from the total line length to see how much room is left. It's simple. Tedious, but simple. You don't even need a computer to do it, all you need is the width scheme for the font. How do you get the width scheme? The width of each character is included in the H-P font files, and I'll show how to extract and use this information further on in this series. The Digi-Font program, which I highly recommend, includes an option to print a width chart.

PCL commands are sent to the printer using a series of escape sequences. The four lines of code in Figure 1 print a 3 x 5 inch black rectangle. The first line positions the dot position at 300,400. The second line sets the rectangle width to 900 dots (3 inches). The third line sets the rectangle height to 1500 dots (5 inches). The last line prints the rectangle with solid fill. The code is shown in human readable form, but is sent to the printer with no carriage returns or line feeds, and with the 1B hex escape character where <ESC> is shown. This can be sent to the LaserJet from any computer with an RS-232 or Centronics port.

```
<ESC>*p300x400Y
<ESC>*c900A
<ESC>*c1500B
<ESC>*c0P
```

Figure 1: PCL commands to set 3 x 5 black box.

The format in Figure 1 is much easier to read than a DEBUG dump, so I wrote the short C program in Figure 2 to convert PCL files to human readable form. This translates the escape character to <ESC> and places it at the beginning of a new line. My LaserJet configuration required the \r return code, but other printer configurations may not require it--my Epson MX-80 doesn't because I have it set to add the return to a newline. This

```
/* SETESC.C -- A program to print the 1B hex escape character code
   as <ESC> for illustrating PCL code examples.
   VERSION 1.0 5/3/89 Compiled with Turbo C V2.0   */

#include <stdio.h>
int c;
FILE *ifd;

main(argc,argv)
char **argv;
{
   char *fname;
   fname = argv[1];
   if ((ifd = fopen(fname, "r")) == NULL)
      {
      printf("Can't open file %s\n",fname);
      exit();
      }
   while ((c=fgetc(ifd)) !=EOF)
   {
      if(c == 0x1b)  /* Escape Character */

         fprintf(stdprn,"\r\n<ESC>");
      else
         fprintf(stdprn,"%c",c);
```

Figure 2: A program to convert PCL code to human readable form.

is **NOT** an example of good C coding practice. It's just a short routine which I wrote late at night when I was anxious to study some PCL routines. I need to write a program to convert human readable commented source PCL files to executable run files.

Why program in PCL when there are so many programs already available? There are a number of reasons. For one, it is not efficient (or even possible) to do everything through wordprocessor or page preparation programs. Another is that you may not be able to locate programs for your machine (such as CP/M, Hawthorne's 68000, or the NS320XX). Another is the high cost of programs which do more than you need. Lastly, programmers have to learn PCL in order to write the programs used by those who can't program. We should also mention the challenge of having to do it in order to understand what is going on.

We must learn about lasers because they will very rapidly replace daisy wheel and dot matrix printers for everything except a few specialized applications such as Cheshire mailing labels and multi-part POS (Point Of Sale) forms. Even the use of mailing labels will drop because people are using ink jet printers to print the addresses directly on the envelope.

I am using PageMaker 3.0, and the more I use it, the more I like it. I wouldn't even consider writing a program to duplicate PageMaker. But, I have some requirements which existing programs don't fill. So, I'm starting on a long term laser programming project.

Soft Fonts--The Potentials and the Problems

You have to use soft fonts in order to realize the Laserjet's real potential, but managing soft fonts presents significant problems for most users.

A soft font is contained in a file which consists of a header plus bit mapped data for each character in the font for one style and one size. A document which uses Helvetica in nine point regular, nine point bold, and ten point bold, will require three font files. The required font files can be purchased on disk, or can be generated from outline data. I prefer to generate the fonts using Digi-Font's program for many reasons which will become apparent as we get into actual application examples.

Soft fonts can contain special characters in addition to the alpha/numeric and punctuation characters in the standard ASCII set. These extended character sets, called symbol sets, contain things such as copyright and trademark characters, bullets and ballot boxes, fractions, accented characters, and many other special characters which typographers are accustomed to (printers call special ornamental characters which are not in the standard sort *dingbats*). There are many different symbol sets available, and there is often a problem because no one set contains all the symbols for an application. I'll show how to customize symbol sets and special

characters in a future article, but see the Digi-Font review in this issue to see how easily it can be done with their program.

The problem with soft fonts is that they take up space, both on the computer disk and in the printer's memory. It also takes time to upload them to the printer. Since the fonts are bit mapped (there is one bit set for every dot which prints plus an unset bit for every dot which does not print) and there are 300 dots per inch, soft font files can get very large. A nine point Palatino font with a small symbol set is 12.1 K. The same font in 48 point is 182.6 K. The ability to generate fonts containing only the required characters and symbols is very important in order to control the font file size. It is also important to be able to easily generate fonts in the required size and then discard them at the end of the job--otherwise you end up with many megabytes of seldom used fonts. The fonts can always be re-generated if needed.

The soft fonts can be uploaded to the LaserJet as either permanent fonts or as temporary fonts. Permanent fonts remain until the printer is turned off or they are deleted under program control. Temporary fonts are removed whenever the printer is reset. The printer does not communicate back to the computer, so there is no way to find out what fonts are loaded, therefore most software programs reset the printer at the beginning and end of every job to remove any temporary fonts. They then reload the fonts they need--even if they are the fonts which were just removed.

The LaserJet has acquired an undeserved reputation for slowness because of the constant reloading of soft fonts, and font management is one of the aspects we will cover in this series. For my own uses, I am planning on setting up a system as a server (even a cheap CP/M or 8088 one will do) for font management. All output for the covered applications will go through the server which will keep track of which fonts are already in the printer and only upload fonts as required. This could also be accomplished as a separate task on a true multitasking system.

LaserJet Applications

The LaserJet offers a unique combination of quality, speed, resolution, versatility, and affordability.

It can crank out draft copies of manuscripts much faster than a dot matrix printer--and they're much easier to read too!. Then it can set the document in a wide selection of proportionally spaced fonts in various sizes from 6 to 720 points. The file size for a font with only the one character "A" in Palatino 720 points is 565 K! The character is also 10 inches high, so only one character will fit on an 8.5 x 11 inch sheet. Large type sizes must be used with care because of the tremendous file sizes, but business and publishing don't require many large characters. When they are needed, partial character sets can be generated and loaded directly to the printer for one time use.

Most drawing and CAD (Computer Aided Drafting) programs, such as Generic CADD which I use, can generate bit mapped output for the LaserJet. Another good program is H-P's Draw Gallery. These programs perform sizing and scaling before they generate the bit map because you get a bad case of the jaggies if the scaling is done from a bit map. There are also routines which convert the output from Borland's Turbo C functions to LaserJet output.

Businesses are using LaserJets to output order forms and invoices which include drawings of the items and bar codes for order processing. The use of bar codes is expanding very rapidly, and it takes a laser printer to provide sufficient resolution for reliable scanning. LaserJets are also used for manuals, catalogs, literature, stock bin labels (with bar codes), forms, etc.

The alternative to the LaserJet is the PostScript devices which contain scalable fonts. This avoids much of the problem with PCL bit mapped fonts, but the printers cost about $2,000 more because of the cost of the PostScript interpreter which is contained in the printer. PostScript is attractive for graphics work using many different type styles and sizes with rotated and scaled images in a single document--a poster is a good example of this. But, I consider this graphic artwork. My main concern is text with some line drawing illustrations such as flow charts, and I prefer the LaserJet. Most of the high volume business use is also concerned with text and simple line drawings.

I consider the LaserJet as a replacement for typewriters, daisywheel printers, dot matrix printers, and phototypesetters, plus the kind of drawing made with a T-square. I consider PostScript as a replacement for the artist's pen and brush. It isn't really that cut-and-dried, there is a lot of overlap, and any job can probably be forced from either system. But, they each have their area of optimum economic performance.

The laser market is changing rapidly. There are packages which enable the MAC to output to the LaserJet, and there are packages for the PC which translate PostScript to PCL. It is no longer an *either-or* choice. During the next few years laser devices will probably evolve to the point where they all accept one standard language. Till then, I'll stick with PCL.

Where Do We Go From Here?

This article, the first in a series, is a brief introduction to the LaserJet. In future articles we will demonstrate the PCL language, develop runoff programs including word wrap and justification of proportionally spaced type, and develop graphic routines, plus forms management. We will also examine the LaserJet font file structure, show how to extract (and how to change) the width values, how to create custom fonts and how to manage the font files, and how to create composite characters and special symbol sets.

We'll also cover typography fundamentals, unusual uses and applications, and hardware and software enhancements for the LaserJet.

This is a lot of ground to cover, and your feedback is welcome. Let us know about your questions, problems, successes and failures with laser printers. ●

The Z-System Corner

by Jay Sage

For this issue I will discuss some unique new capabilities made possible by NZCOM that have already proved their value and that I hope will be exploited to a much greater extent in the future. I originally had several other issues on my agenda, but Lee A. Hart sent me such an interesting article that I wanted to leave plenty of room for it.

System Enhancements Using NZCOM

I'm afraid that many people think of NZCOM as just a way for unskilled users to get Z-System running on their computers. It's true that NZCOM accomplishes that, but, as I have asserted often before, NZCOM does much more than that. It offers possibilities that a manually installed Z-System cannot achieve. I would like to describe one of them here and will do so first in the context of a problem that arose with the new ZSDOS and ZDDOS disk operating systems on some computers.

Because CP/M was created originally for the 8080 microprocessor, a number of BIOS implementors (the BIOS, or Basic Input/Output System, is the hardware-dependent part of CP/M) felt that they could make free use of the additional registers introduced with the Z80 chip. These registers included two index registers, called IX and IY, and a duplicate set of the standard registers denoted with primes on the names (e.g., B' or HL').

This was perhaps excusable at the time, but it is poor programming practice for an operating system to change anything other than what is explicitly indicated in the specifications. Most BIOS writers who have used the Zilog registers have been careful to restore the original values before exit from the BIOS routines. Unfortunately, a few BIOSes fail to do that. Among them are the following: Epson QX10, Zorba, Televideo 803 and TPC-1, Oneac On, and the Osborne Executive. The Bondwell is on the suspect list, and there are probably others we don't know about yet.

The (mis)use of these registers poses no problem for programs written to run on the 8080, but today we are rapidly moving beyond the limitations of the 8080 and making extensive use of the Z80 registers to pack more power into operating system components and application programs. Today, the Z-System, true to its name, is intended to run only on the Z80 or upwardly compatible processors like the HD64180, Z180, or Z280.

Several users who purchased ZDOS (that is ZSDOS and ZDDOS) found that it would not work properly on their computers. An investigation turned up the fact that the BIOSes in those computers were modifying the index registers. This also explained why those same users had been experiencing strange problems with JetLDR, Bridger Mitchell's superb Z-System module loader. It also explained some mysterious problems I was having with a number of programs (for example, EDITNDR) on my Televideo 803! The question was what to do about the problem.

In the ancient days, when computers always came with the source to their BIOS and their owners were always intimately familiar with the procedures for rebuilding their operating systems, the solution would have been to rewrite the BIOS with the proper PUSH IX and POP IX instructions to preserve the index register values. But what could we do today for nonprogrammers and those without BIOS source code? NZCOM provided the answer quite nicely!

As I explained in a column long ago, NZCOM works by creating what I call a virtual BIOS (I'll call it VBIOS) lower in memory in order to open up space for the Z-System modules between it and the real BIOS (often called the custom BIOS or CBIOS). The source for this virtual BIOS is available. Except for the warmboot code and some minor complications for IOP (input/output processor) support, the standard NZCOM VBIOS module just vectors calls that come to it up to the real BIOS.

But no one says this is all it is allowed to do. ZDOS authors Cam Cotrill and Hal Bower found it quite easy to surround the vectors with code to save and restore registers. First they released ZSNZBI11.LBR, which contained the source and ZRL file (loadable by NZCOM) for a VBIOS that preserved the IX and IY registers for all disk function calls. Later they discovered that some of the machines changed the index registers even for console I/O function calls, and others changed the alternate registers. They then wrote ZSNZBI12.LBR, whose VBIOS preserves all the registers for all BIOS functions.

Instead of having the virtual BIOS routines jump directly to the real BIOS, they jump to an intermediate entry point. For example, the list-status vector in the jump table has a JP ILSTST (intermediate list status), and the code at ILSTST is:

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER file maintenance shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (on PC-Pursuit). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Finally, Jay recently became the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail.

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog computation to solve problems in signal, image, and information processing.

```
ILSTST: LD      A,45
        JR      DOBIOS
```

The offset for the BIOS function is placed in the A register and then control is transferred to a general BIOS-calling routine shown in Table 1 that implements the register protection. The routine JPHL referenced there contains only the code line JP (HL), which vectors the CPU off to the BIOS with a return to the DOBIOS code.

To use this replacement VBIOS, you have to run MKZCM and create an NZCOM system with 4 records allocated for the BIOS instead of the standard 2. Because the BIOS must start on a page rather than just a record boundary, MKZCM will sometimes make automatic adjustments to the BIOS size. Therefore, you should specify changes in MKZCM starting with the higher-numbered modules; the adjustment in the BIOS allocation should be made last. If an attempt to enter a value of 4 results in MKZCM using 5, then you could go back (if you don't like wasting memory) and make one of the other modules (such as the NDR) one record larger and then respecify a 4-record BIOS.

There are many other ways that NZCOM can be used to introduce system enhancements without having to make changes in the real BIOS. As an example, we will show how to add support for the drive vector in environment descriptors of type 80H and above (implemented with NZCOM and Z3PLUS). The drive vector is a 16-bit value stored as a word beginning at offset 34H in the environment descriptor. It specifies which disk drives are actually implemented on a system. The lowest order bit in the word is for drive A and the highest for drive P. A zero in a bit position indicates that the corresponding drive is not available. For example, on my SB180 the bytes at addresses ENV+34H and ENV+35H were 7FH and 00H, respectively. Thus the word value is 007FH or 0000,0000,0111,1111 binary, indicating that I had drives A, B, C, D, E, F, and G. When the hard disk E partition developed a problem that made it unusable, I changed the 7FH value to 6FH, thereby disabling drive E. You can display the drive vector using menu selection 3 in the SHOW program (ZSHOW for Z3PLUS).

The ZCPR34 command processor knows about the drive vector and will not allow command references to unsupported drives. But what about programs that you run? Unfortunately, the BIOS generally knows only which drives are **potentially** implemented, and it may try to access a nonexistent drive. When 'smart' BIOSes encounter this problem, they often prompt the user as to what to do next. This is fine if you are sitting at the console and can take appropriate action to recover, but if the system is being run remotely, there is generally no way for the remote user to recover. In fact, because the 'smart' BIOS uses direct hardware calls to display the "Abort, Retry, Ignore?" message rather than calls through the BIOS vector table, the remote user does not even see the message and just thinks the system has crashed. My Z-Node got hung once that way when I forgot to put a diskette back into a floppy drive. A caller attempted to access it, and when I got home, the BIOS was dutifully

```
DOBIOS: LD      HL,CBIOS   ; Start with address of real BIOS
        ADD     A,L        ; Add offset in A (never a
        LD      L,A                ; ..carry since on page boundary)
        EXX                        ; Swap to alternate registers
        LD      (HLP),HL   ; Save them all in memory
        LD      (DEP),DE
        LD      (BCP),BC
        LD      (IXREG),IX ; Save index registers, too
        LD      (IYREG),IY
        EXX      ; Back to regular registers
        EX      AF,AF'             ; Swap to alternate PSW
        PUSH    AF         ; Save it on stack
        EX      AF,AF'               ; Back to original PSW
        CALL    JPHL               ; Actually call the BIOS!
        EXX      ; Restore alternate and index
        LD      HL,(HLP)   ; ..registers
        LD      DE,(DEP)
        LD      BC,(BCP)
        LD      IX,(IXREG)
        EX      AF,AF'               ; Alternate PSW, too
        POP     AF
        EX      AF,AF'
        RET

; Register save area

BCP:    DEFS    2                  ; BC'
DEP:    DEFS    2                  ; DE'
HLP:    DEFS    2                  ; HL'
IXREG:  DEFS    2                  ; IX
IYREG:  DEFS    2                  ; IY

        END
```

Table 1. Code from ZSNZBI12.Z80 that preserves all index and alternate registers across BIOS calls in NZCOM.

```
; Add DRVEC definition in the ENV common

   COMMON   /_ENV_/
Z3ENV:
DRVEC EQU   Z3ENV+34H      ; Drive vector
CCP   EQU   Z3ENV+3FH
DOS   EQU   Z3ENV+42H

; Modify ISELDK as follows and place it after the DOBIOS code and before
; the data area for register storage.

ISELDK: LD HL,(DRVEC)      ; Get drive vector
   LD A,16      ; Subtract requested drive
   SUB   C      ; .. from 16
   LD B,A       ; .. and put into B
ISELDK1:
   ADD   HL,HL     ; Move bits left into carry
   DJNZ  ISELDK1      ; Loop 16-<drive> times
   LD HL,0      ; BIOS return code for invalid drive
   RET   NC     ; Return if drive vector bit not set
   LD A,27      ; Otherwise, use CBIOS function
   JR DOBIOS       ; .. at offset 27
```

Table 2. Code added to virtual BIOS to support the environment drive vector at the BIOS level.

beeping at me and waiting for me to tell it what to do.

My first stab at writing a VBIOS that observes the drive vector restrictions is shown in Table 2. Now that I have read Lee Hart's column, I am sure that this code can be made shorter, faster, or both! But I will leave that as an exercise for the reader. (I already see one place where I could save a byte.)

The listing assumes you are starting with the ZSNZBIO described earlier. Just add the extra equate for DRVEC under the /_ENV_/ common block and replace the simple ISELDK (intermediate select disk) code with the slightly more complex version shown in the Table. I put this on my Televideo, and the results were most pleasant. Now when I attempt to access a nonexistent drive, the system does not force a direct-CBIOS warmboot that drops me out of NZCOM.

These two examples of system enhancements by no means exhaust the possibilities. One can implement all kinds of additional features and drivers right in the NZCOM VBIOS. Joe Wright suggested early during NZCOM development that one should create an absolutely stripped down CBIOS, one that contains only the functions that are absolutely necessary to get the system running and then implement all the bells and whistles in the VBIOS. These extra features would include things like RAM-disk drivers, keyboard type-ahead buffers, logical drive swapping facilities, and disk error recovery management routines. With this strategy, one can actually achieve a larger TPA with an NZCOM system than one had under the standard CP/M system, since the CBIOS can be made smaller and the fancy features dropped when a larger TPA is more important.

For example, the "Abort, Retry, Ignore" message should be implemented in the VBIOS, with the CBIOS returning from disk errors with standard error codes. With the normal VBIOS, the error will simply be passed back to the DOS, which will report the error in its usual way ("BDOS ERROR on . . ." in the case of the Digital Research BDOS). A more elaborate VBIOS can detect the error, report it to the user, and allow the operation to be retried. When the system is running in remote mode, either the simpler VBIOS can be used or the prompt can be vectored properly through the jump table so that the remote user will be able to deal with the problem.

Similarly, one should be able to handle the swapping of logical drive names in the VBIOS. There are a couple of pitfalls to watch out for, however. If you change logical names, you better make sure that the disk system is reset, probably both before and after the swap. You also better make sure that NZCOM can still find its CCP file, which is normally kept in directory A15:. If you swap the A drive without providing a copy of this CCP in the new A drive, you'll be in serious trouble. Of course, the swapping would be handled by a utility program, and it would worry about these requirements. The VBIOS would simply have the code for translating references to a logical drive value in register C into a possibly different physical drive value.

I hope that this short discussion has given some of you ideas for imaginative applications of the new capability offered by the NZCOM virtual BIOS. If so, I would love to hear about them and to see sample code. •

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 - $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)

Generating LaserJet Fonts

A Review of Digi-Fonts Version 2.0

by Art Carlson

I am replacing my Compugraphic phototypesetter with a LaserJet, and I have been looking for a type generation system which runs on my AT clone. Since I am coming from a professional typesetting background, my expectations are much more rigorous than someone whose experience has been with daisy wheel or dot matrix printers. I expect the laser system to do everything my phototypesetter does, plus do it better, faster, and more conveniently. That's asking a lot, and most DTP products have not yet matured to the point where they meet my expectations.

Since the ability to generate type fonts is the key to laser based typesetting, I have been spending a lot of time and effort looking for a suitable type generation system. Several products managed to produce type fonts, but they didn't fully satisfy my demanding requirements. When I tried Digi-Fonts it did most of what I wanted, and when they released version 2.0 there was no longer any question about what type generation program I would be using.

Digi-Fonts

The Digi-Fonts system consists of three parts. The first is *Digi-duit!* which consists of programs and utilities which do the actual work. Second is the typeface library (now 34 disks and growing). Third is *Digi-install!* which installs the fonts and creates matching screen fonts if needed. Actually, there is a fourth element which is equally important--that's the manual which provides a lot of good technical information.

I have chosen Digi-Fonts because it provides the tools and information which enable me to customize the fonts to do exactly what I want. This is very important to me. I don't like to be forced to make do with what someone else decided that they want. As I go into the details of some of the things that I can do with Digi-Fonts, keep in mind that it doesn't have to be this complicated for normal applications. Digi-Fonts is easy to use in the average office. You don't need to do the things I talk about unless you want to provide solutions to the problems other programs leave unsolved. I view Digi-Fonts as a toolkit which makes average applications easy, and advanced applications possible. You only have to use as much of the power as you need.

Digi-duit 2.0 sells for $89.95 plus $4 shipping. It includes disk 00 with eight DIGI-FONTS (similar to Palatino, Palatino Bold, Palatino Italic, Palatino Bold Italic, Commercial Symbols, Bauhaus Medium, Park Avenue Script, and Hobo). Also included are an extensive manual and a catalog of the typeface library which now has 33 disks. Additional disks are currently sold separately (eventually they will become part of Typeface Library Vol. 2). For a little over $90 you'll be able to do everything which I demonstrate in this article (plus some things which I'll probably forget to mention). Digi-install versions for WordPerfect 5.0, Ventura/GEM, PageMaker/Windows, or MicroSoft Word 4 & 5 are available for $20 each.

The fonts come in outline form, and you can generate the bit-mapped fonts you need when you need them. You can store the fonts which you use frequently, and regenerate the seldom used ones when you need them instead of tying up disk space. DFI even provides a function to save the commands needed to create the fonts for a particular job to a batch file that you can run whenever you need the fonts.

Character Modifications

It used to be that a character was a character and if you wanted something different you had to buy another font. But, now with electronic wizardry, you can create many different faces from one font file.

With DFI you can make fonts from 3 to 720 points in 1/10 point increments (there are 72 points to an inch, so 720 points equals 10 inches). If a title is a little too long, or if the text doesn't quite fit the page, you can change the type size to make it fit. You can also alter the type size to provide the correct emphasis and balance between the elements.

You can also modify the width of the font as shown in Figure 1. The default width is equal to the point size, but you can change it in 1/10 point increments. This is another way to make type fit the job or to add emphasis. How expanding the word "Stretch" to twice its normal width for an ad about stretching your dollar? Or reducing the width for an ad about being squeezed for space? The use of partial character sets will be very useful for these applications.

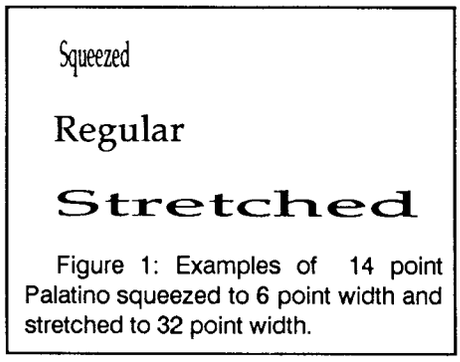


Figure 1: Examples of 14 point Palatino squeezed to 6 point width and stretched to 32 point width.

You can select between proportional and fixed character width to make the characters in a normally proportionally spaced font line up in columns. They may look a little odd, but they will line up. If you frequently have this requirement, DFI has fixed width fonts in its library which I'll be changing to for program code listings.

Fonts can be flopped, slanted, or rotated as shown in Figure 2. Flopped fonts are a mirror image, and an example is the word *Ambulance* painted on ambulances.

It looks backwards until you look at it through your rear view mirror. Slanted fonts look something like italic. Their feet are on the baseline, and the top is pushed over to the side. You can slant type within the range of -45 to +45 degrees. In rotated type the entire character is turned instead of being distorted as in slanting. You can rotate type in the range of zero to 360 degrees. An application for rotated type would be in preparing a form on the LaserJet II which can not use both portrait and landscape fonts on the same page. Standard wordprocessor or page preparation programs don't know how to handle flopped or rotated fonts. You'll have to use tricks or use a special program.

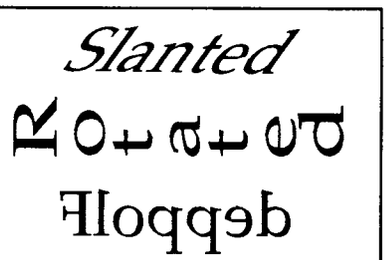


Figure 2: Examples of slanted, rotated, and flopped type.

Fonts can also be reversed into a white character surrounded by a black box as shown in Figure 3. Avoid reversed type in the smaller sizes or with fonts having very fine lines, as it may block up in the printing process and and be difficult to read. Small areas of reversed type output on a clear transparency sheet can sometimes be substituted for a camera negative.

Figure 3: An example of reversed type.

The font can be made as an outline, with the portion enclosed by the outline filled with white or a pattern. Normal black type is merely the outline filled with black. You can also turn the outline off and just print the shaded portion. DFI provides a number of gray scale and design patterns from which you can choose. The patterns are generated from simple text files, and DFI provides instructions for writing your own pattern files. You shouldn't have much of a problem creating your own patterns after reading the instructions and examining the pattern files which they provide.

You can make shadow fonts with control over the direction and the length of the shadow. The shadow can be printed in one of the patterns.

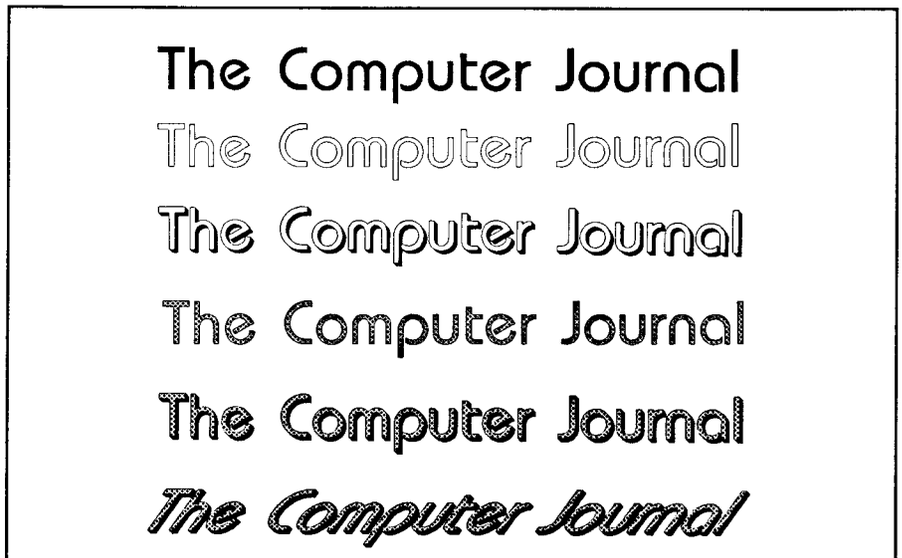


Figure 4: Examples of outlines, drop shadows, and shaded type.

And, of course, you can generate fonts in either portrait or landscape.

These effects can be combined to produce some very interesting effects. For example, a gray filled outline font with a black shadow. Or a reverse font with the surrounding portion printed in a pattern. Figure 4 illustrates a few of the possibilities.

The ability to make partial character sets is a real benefit for these special effects where you usually only require a few characters.

Partial Character Sets

One of the big disadvantages of bitmapped fonts is that they can take up a lot of space on the disk and in the printer. Large font files also take longer to upload to the printer. In order to alleviate this problem DFI enables you to specify which characters you want in the font. This is closely connected with symbol set which I'll discuss next. Right now I'm talking about the primary alpha/numeric characters.

As an example, I ran some tests using 36 point Bauhaus Medium normal--all tests were performed on an 12.5 MHz AT clone with a 40 Meg hard drive. I first generated the full ANSI Windows set, which took 1:58 (minutes:seconds) with a file size of 148,378. I then generated just the ASCII set which took 0:57 with a file size of 70,591. Last, I generated only the characters for the words "The Computer Journal". *(Editor note: I know that the period is always supposed to go inside the quote marks, but that would indicate that I also generated the period, which I didn't. When quotes are used to mark a command or actual characters, I place the period where it does not create an error!)* This took only nine seconds and the file size was 9,860.

With DFI you can use Hex, decimal, or ASCII characters to specify the characters you want. I just entered "The Computer Journal". DFI put them in order and only generated the ones used. This feature is extremely important for the larger fonts which require more time and space. It's even more important when using the special effects which require even more time. It is very convenient when experimenting with special effects because you can generate just a few characters in order to see how it looks.

Symbol Sets

The ability to print special characters in addition to the usual ASCII numerals, alpha characters, and punctuation, is one of the very important advantages of the laser printer. Providing the special characters (called symbol sets) for a specific job is one of the headaches.

The characters are represented by one byte per character, which limits the number of different codes to 255 (FF Hex). The first 32 numbers (00 Hex to 1F Hex) are reserved for special control codes in normal ASCII, although some programs also use these for special characters. ASCII only uses the lower seven bits (00 Hex to 7F Hex) which is 128 numbers. Subtracting the 32 numbers reserved for con-

trol codes leaves 96 numbers to represent the standard ASCII character set.

Special characters (called extended characters) make use of the full eight bits. This allows for up to 128 additional characters, although some programs reserve the high bit set control codes (80 Hex to 9F Hex) for control codes because many printers respond to numbers in this range as control codes. It is best to assume that there are 96 additional characters available.

The challenge is to fit all the extended characters everyone needs in 96 spaces. It can't be done. The answer has been to create many different symbol sets. The problem is that all the extended characters required for a job may not be available in any one symbol set.

People got along fine for years with just the 96 character ASCII set. Why do they now need more than the 192 characters which would fit in an extended character set. It is because typists made do with a limited set, but typographers and printers had a very rich and extensive set of characters. Before lasers, typists used their limited set to prepare rough copy and typographers added the special characters. Typists typed, and typographers prepared typeset copy. Now, people who may have only the skills of a typist, expect to sit down at a computer and produce beautiful typeset copy--just like what they used to get from the typographer. In order to accomplish this they have to use the special characters which are one of the things which make the difference between something which is typewritten and something which is typeset.

Some of the more frequently used symbols are the diacritical marks such as the cedilla, circumflex, macron or umlaut, used in foreign names; copyright, registered, and trademark; the degree symbol used with temperatures; the EM and EN dash; bullets; ballot boxes; the cent sign for pennies; and math symbols such as plus/minus, divide, or multiply (typographers do not use the character X for multiply).

I am currently working on a manuscript which requires the diacritical marks, the degree and copyright symbols, slashed zeros, bullets, black boxes, and math symbols. The only way to handle this without DFI was to generate a different complete font in each size and face for every one of the required symbol sets. This would involve a lot of fonts and disk space. It would also require complicated style templates and a lot of embedded tags to force it into PageMaker. I thought DTP was supposed to make it easier!

Now with DFI the solution is simple because it allows me to select the symbols to make a font with exactly what I need (as long as I don't need more than 96 extended characters).

The symbol set is specified in a file with the SET extension, and consists of the following commands:

1) **Transfer**--to move one or more character codes from the source font to the printer font.

2) **Composite**--to create a new printer character from one or more source characters.

3) **Parameter**--to change the basic scaling parameters of the font: pointsize, pointwidth, slant, rotation, and flop.

4) **Font**--to switch the source font to a different Digi-Fonts font from which the following characters will be drawn.

Using the Transfer and Font commands I can start with a basic font such as Palatino, transfer the degree sign from the Palatino DFI set, and transfer the bullet, black box, and math symbols from the Commercial Symbols font. The composite command allows you to specify the position and scaling size of the individual elements, and is used to make the slashed zero and accented characters.

In Conclusion

Several things impressed me about Digi-Fonts. The first was that their manual explains the file structure and contains enough technical information to allow you to modify the files if you so desire. They even include some information on coding outline data which may be enough to get you started if you want to create a few custom characters which can be used with their scaler. The scaler can be run independently of the menu, and can even be executed from an application program. I like the fact that they provide the technical information which is missing from so many other manuals.

The second thing is that they have a very powerful product. It can be used for the easy jobs, but it is also a toolkit which enables me to do the formally impossible jobs.

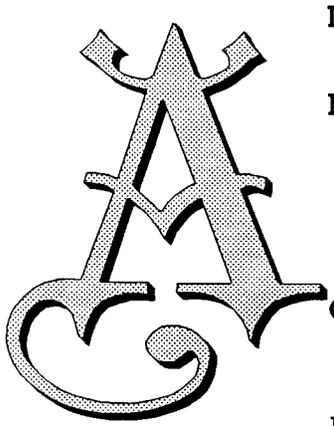
Third the product is very reasonably priced with a large library of fonts available.

Digi-Fonts is not standing still. They are preparing additional fonts such as fractions, which are not handled well by any other programs. This will be a symbol set. That is, you will be able to make a fraction font using any typeface style, by merely selecting the fractions symbolset. It will be included on disk 35, along with Courier, Line Draw, Line Printer, and others. Several other vendors will be using DFI's outline font library with their products. Corel Draw, which will be available July 1989, 1.1 yses Digi-Fonts outlines directly.

I strongly recommend Digi-Fonts for anyone with access to a LaserJet compatible printer. I will be using their product to demonstrate applications in the PCL programming series. •

Billions of Fonts!

DIGI-FONTS for HP LaserJet Plus or II printers



Largest Selection
More than 272 styles

Most Flexibility
Digi-duit! 2.0 makes high quality fonts any size, 3-720 pts, plus slant, flop, rotate, reverse & more! Incredible Special Effects.

Quick and easy
Works great with most software programs.

Lowest Prices

Digi-duit! 2.0 — **$89.95** + $4 ship
8 DIGI-FONTS, Typeface catalog

Complete Set — **$489.90** + $7 ship
Digi-duit! 2.0 + 272 DIGI-FONTS

Call Today!
1(800) 242-5665

DIGI-FONTS, INC.
528 Commons Drive
Golden, Colorado 80401
(303) 526-9435 FAX (303) 526-9502

This ad set entirely with DIGI-FONTS Typefaces. VISA, MC, AMEX, COD

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

JetLDR **$20**

Z-System segment loader for ZRL and absolute files. (included with Z3PLUS and NZ-COM)

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **JetLDR** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

Advanced CP/M

Making Old Programs Z-System Aware, Z-80 Interrupt Bug

by Bridger Mitchell

How can a programmer add Z-System capability to a program she or he is writing in a high-level language (HLL) such as C or Pascal? David Goodman raised this interesting question earlier this year. The latest version of the BDS-C compiler as well as Turbo Modula-2 have built-in access to the Z-System. But most HLL compilers were written before the Z-System was fully developed, and do not provide for obtaining the address of the Z-System external environment or using its features.

A similar situation arises occasionally, when one wants extend a well-functioning program with additional Z-System features, but source code for the program is unavailable. New routines can sometimes be patched in after a limited disassembly to determine key spots in the flow of program control.

The Standard Z-System Header

In both cases the common need is to obtain the address of the Z-System environment. A Z-System application begins with this standard header (the addresses shown are those when the application is loaded at 0100h):

```
0100: jp  start
0103: db  'Z3ENV',1 ;signature for ZCPR command processor
0109: dw  0000      ; where ZCPR puts z3env address
```

When the ZCPR 3.3 or 3.4 command processor loads a COM file to 100h and finds the 'Z3ENV' signature at 103h it "installs" the external environment address at 109h. Thus, the application starts up with the running system's environment address available to it at 109h. Of course, the programs that don't know about the Z-System have other code at 100h.

A newcomer to CP/M might well ask, Why is such a round-about approach used? Why isn't the external environment address available from a system call, or at a fixed, absolute address? Had the Z-System been developed as part of CP/M, the environment address would most likely have been directly supplied via a BIOS or BDOS system call. But coming on the scene much later, and striving to be compatible with existing systems, the Z-System required special code in both the ZCPR command processor and in the Z-System-aware application. In addition, expanded Z-System headers have been developed for "type-3" and "type-4" applications. In these cases, the command processor and the application are tightly coupled during the program loading process, and the application is usually loaded, or relocated to high memory and executed there.

Actually, there's a good chance that an absolute address, somewhere in the first 64 bytes of the CP/M memory map, would have served nicely as a permanent location for a pointer to the Z-system environment. And the retrofit solution I develop in this issue's column uses such an address.

In any case, the standard is now based on the 11-byte Z-System header just described. The difficulty is that, in an HLL that antedates the Z-System, the very first bytes of a compiled program often contain the key run-time routines needed by most every program developed with the compiler. And in assembly-language programs those first bytes can be almost anything.

You might initially think that it would work just to move the code (or code and data) in those first bytes to the end of the program and replace them with the standard Z-System header. You would make the "start" address the now-moved code, and at the end of that you would add a jump back to the address following the header. However, this quickly becomes a messy, case-by-case patch. Moreover, in many instances, other routines in the programs may expect to find parameters or entry points at particular locations at the head of the program. If you're lucky, the patched program simply won't run; otherwise, it will do something, but get bad values or behave incorrectly.

In the remainder of this column I discuss two interesting ways to add Z-awareness to older programs.

The Slider

In what I will call the "slider" approach we take the COM image of the program and *prepend* both a Z-system header and some preamble code in front of it. The extra code first saves the environment address in a safe place, and then moves the COM image down in memory on top of itself. Finally, it begins executing the original program.

The SLIDER code in Figure 1 is a bit tricky. The basic idea is to use the block move LDIR instruction to slide the original program down in memory, and on top of, the very code that does the move.

SLIDER didn't take this final form immediately. At one point I momentarily convinced myself I had discovered an elegant way to do the slide; it involved putting the LDIR instruction just preceding the start of the TPA, and moving the bytes that had originally been there (the tail end of the command line buffer) back into place as part of the slide-down process. But that was indeed too clever – I had forgotten that the Z80 actually executes its auto-repeated instructions by repeat-

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); Back-Grounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

Figure 1.
The SLIDER Approach

```
SAFEADDR equ   033h                  ; location of safe 5-byte buffer

; Z-system header, at beginning of program.

   jp pre
   db 'Z3ENV',1   ; signature for ZCPR command processor
envptr:  dw 0000           ; where ZCPR puts z3env address
;
; code moved to SAFEADDR+2
;
   ldir
   ret
;
; followed by code to save the environment and
; slide the original program to 100h

pre:  ld hl,envptr         ; save environment pointer from header
   ld de,SAFEADDR ; and move "LDIR  RET" code too.
   ld bc,5
   ldir
;
gettop:  ld hl,0           ; get SP value to HL
   add   hl,sp
   push  hl    ; save on stack
   ld de,(0006)   ; if SP < top of tpa
   or a,a
   sbc   hl,de
   pop   hl    ; ..recover SP
   jr c,getsize   ; ..use SP
   ex de,hl    ; ..else use top of tpa
getsize:ld  de,origcode    ; calculate bytes to move
   or a
   sbc   hl,de
   ld b,h      ; into bc
   ld c,l
   ex de,hl    ; source -> origcode
   ld de,100h     ; destination -> tpa
   push  de    ; put startup addr on stack
   jp SAFEADDR+2  ; jump to LDIR
;
;------------------
origcode equ $             ; load original COM file to here
```

Figure 2

The ROLLER Approach -- Preamble

```
   org   100h
;
   jp 0000     ; patched later by debugger
   RET         ; return opcode at 103h
```

edly fetching the opcode from memory for each execution of the repeat "loop". Thus, as soon as any byte was moved on top of the LDIR the next fetch would get a totally different "opcode" and everything would crash.

Consequently, the final SLIDER code first copies both the environment address that has been installed by the command processor in the header and three additional bytes of code to the safe address on page 0. After calculating how much to move, SLIDER will jump to these code bytes (LDIR, RET). They will be the last SLIDER instructions executed, with the return instruction sending control to the original code at the beginning of the TPA (0100h).

How many bytes should be slid? Well, we could determine this for the specific program, and patch in the exact value each time SLIDER is prepended to an old program. Instead, I've chosen to make the calculation automatic, and move the maximum number of bytes (the extra milliseconds required should be of no consequence in this application). Usually, it would be prudent to slide the entire TPA. But it is possible that the active stack is internal to the command processor (rather than an external stack in a Z-System buffer). Moving the contents of the stack would create havoc for an application that exited by returning to the command processor rather than with a warm boot. So, if the stack pointer address is in the TPA, the SLIDER code moves only the bytes below that point.

After it has been run once, the in-memory image of the original program can be reexecuted by the command-processor's GO (or JUMP 100) command (this assumes, of course, that the original program is itself reexecutable -- sometimes it won't be). When reexecuted, nothing gets moved: what will be at 100h is the original image of the COM file (except as changed by its actual execution), with the environment address already in the safe location from the initial execution.

You can see how this works once you've built the SLIDER by using the following GET and PEEK sequence.

```
GET 100 testfile.com
PEEK 100
JUMP 100
PEEK 100
```

The slide-it-down method should be fairly widely applicable. It requires one absolute five-byte buffer outside the TPA -- a place where a copy of the Z-System environment address can be stored for routines in the program to access. In the code, I've used five bytes at 0033h. This is part of the 8-byte region reserved for RST 30 -- the seventh Z80 restart instruction. Although a very few debuggers use this RST, most use RST 38. Because the debuggers install a 3-byte absolute jump instruction (at 0030h or 0038h) and leave the other five bytes unused, those bytes are good candidates. However, some remote CP/M system software uses bytes beginning at 003Bh. At the moment, I believe 0033h is the safest choice.

You might think that another RST location would be an even better choice. Well, probably not. A number of BIOSes use one or more of the lower RST regions for interrupts, clock buffers, and disk parameters. The chance of colliding with *some* BIOS is much higher.

Putting It Together

How do you cobble the SLIDER onto an existing COMFILE.COM program? Here are the steps:

- assemble the slider to binary, name it SLIDER.CIM
- determine the number of records in COMFILE.COM, using e.g. ' ' S D COMFILE.COM /C"
- GET 100 SLIDER.CIM
- GET COMFILE.COM
- SAVE <NN+1> NEWFILE.COM S, where NN+1 is one

  record more than the original size

This can almost be made into an automatic ZEX script; only the number of records to be saved must be entered from the console. Moreover, the SLIDER.CIM

is position-independent and can be reused with another old program. (I use the CIM file type, to emphasize that the image is not an executable file. That way the command processor won't attempt to *run* SLIDER, should you ever mistype it as a command.)

Ok, this gets the dirty work done. Of course, you will need to add your own code to *use* the Z-System environment. You might, for example, access the termcap parameters using HLL routines and incorporate them into screen management functions. All of this code can use the value it finds at 0033h as the pointer to the environment. In effect, the slider method converts an unknown, run-time address into a known absolute address, so that HLL and other subroutines can be compiled and linked with an address that is determined in advance.

If it is a program in COM file format that you are augmenting, you face a more extensive task -- disassembling some of the code to understand the program and find suitable points at which to divert control to your new routines. But the same slider method will serve to obtain the environment address that your Z-System routines will need.

Whatever the specifics of your Z-System routines, be certain that the first one that uses the environment pointer value verifies that it is non-zero. A null value would mean that the command processor did not install the environment address in the header. The host system is therefore not ZCPR 3.3 or later. And an attempt to use 0000 as the environment address will probably garble or crash the program.

Limitations.

The slider approach is reasonably portable. However, I can think of a few areas in which difficulties can arise.

The newly-Z-conscious application could conflict with other programs, such as configuration utilities, that refer to the *disk image* of the application. A specialized program designed to patch default options into the application would expect to find the data bytes to be patched in their original locations in the file. If run on the modified application, the configuration program would either fail to make the patch -- protesting that it can't recognize the file -- or proceed to change the wrong bytes.

A second type of limitation could occur when other, previously-coded patches are added. If they are added to the slider-modified file there will be conflicts. The solution is to do all other patches first.

Finally, under BackGrounder ii it is possible to attach program-specific keyboard macros to an application. BGii does this by appending the macro definitions to the end of the COM file. This will continue to work just fine on a slider-modified file. If your file already has appended definitions, they should be removed before making the slider modifications, and then attached anew.

A Roller Version

I discussed a preliminary version of SLIDER with Jay Sage, and, as always, he had a number of excellent ideas for improving things. Jay outlined a second approach that involves "rolling" off just four bytes at the beginning of the original file, copying them into a buffer appended to the end of the file, and putting the environment-detection code after that.

ROLLER thus consists of a preamble and a postamble,

Figure 3
The ROLLER Approach -- Postamble

```
; Position-independent code.  Assemble to ROLLER.CIM

SAFEADDR equ   033h

   org   100h     ; any origin will do

buf:  ds 4     ; buffer for 1st 4 bytes of orig. program
patch:
   push  hl    ; save potential env addr
   inc   hl    ; skip to 'Z3ENV' signature
   inc   hl
   inc   hl
   ld a,(hl)      ; check 1st 2 bytes
   cp 'Z'
   jr nz,bad
   inc   hl
   ld a,(hl)
   cp '3'
   jr nz,bad
   ld de,1Bh-3 ; point to reflexive address in environment
   add   hl,de
   ld a,(hl)      ; get it
   inc   hl
   ld h,(hl)
   ld a,l
   pop   de    ; get potential env addr
   push  de
   or a,a      ; compare reflexive address
   sbc   hl,de
   ex de,hl    ; potential env addr to HL
   jr z,store     ; Z = found the environment
bad:  ld hl,0     ; store NUL if no environment
store:   ld (SAFEADDR),hl  ; at a known, safe location
   pop   de    ; clear env addr from stack
;
; find Where Am I address
;
   ld hl,-2    ; precompute SP address
   add   hl,sp    ; so we can pop return addr from stack
   call  103h     ; call the RET instruction, pushing wai
wai:  ld sp,hl ; point SP at return addr.
wail: pop   de    ; get return addr from stack
            ; if interrupted during "ld sp,hl"
            ;   de = wail, else de = wai
   ld a,(de)      ; get opcode at return addr
   add   a,10h    ; set CY if HL points at LD SP,HL (0F9h)
            ; clear CY if hl points at POP DE (0D1h)
   ld hl,-(wail-buf)       ; compute abs. address of buf
   adc   hl,de    ; (if ret addr was wai, CY corrects)
;
   ld de,100h     ; restore 1st 4 bytes of program
   push  de
   ld bc,4
   ldir
   ret         ; return to run program
```

shown in Figures 2 and 3. Instead of using a standard Z-System header, it relies on the fact that, in addition to placing the environment address in a standard header, ZCPR 3.3 and 3.4 call the program with that address in the HL register pair. The postamble code, therefore, first verifies that that HL value points to a valid environment, and then stores the pointer at the SAFEADDR location.

Note that it uses the handy trick of calling a routine (at 103h) that simply returns, in order to get the address of the next instruction onto the stack. With that available, it can calculate its own address (wai = "where am I?") and finally the address of the 4 original bytes to be restored to the head of the program.

To make this calculation correctly requires some fancy footwork to handle the possibility that an interrupt occurs during the calculation and changes the address on the stack. I didn't get it right the first time, and I'm grateful to Don Kirkpatrick for suggesting the final code. You will see why he was thinking along these lines further on in the column.

The steps to put ROLLER together are:

- assemble PREAMBLE to binary or hex
- assemble ROLLER to binary, call it ROLLER.CIM
- load debugger and original program, and note "next" address
- move bytes at 100-103 to NEXTADDR
- load PREAMBLE.CIM (or HEX)
- patch the JP address at 101h to NEXTADDR+4
- load ROLLER.CIM to NEXTADDR. Note new next address.
- exit debugger
- save nn pages

Alternatively, you can skip the step of assembling the preamble into a separate file and instead assemble the bytes interactively with the debugger.

Clearly, ROLLER requires a bit more manual effort (though a good ZEX script will ease that, too). But it has the advantage of leaving all but the first four bytes of the program intact in the file image. This approach may be better, then, for any program that has a companion configuration utility.

Interrupt Bug Unmasked

In the previous Advanced CP/M column I discussed situations in which it is necessary to disable interrupts before using the stack pointer for special purposes, or when modifying critical code. I included a routine to test whether Z80 interrupts were enabled before disabling them, and a companion routine to conditionally enable interrupts if they had originally been enabled. But I noted that some peculiarities with the Z80 interrupt status had been reported, and asked readers for help.

Well, only a day or two after my own copy of TCJ #38 arrived in the mail Don Kirkpatrick was on the phone to say that he'd just read his copy and was sending me the straight scoop on detecting interrupt status in the Z80. Don has done a fair amount of testing and established that the Z80 has a subtle defect -- it effectively "lies" and gives a false interrupts-enabled status value in one particular case -- when an interrupt occurs exactly during the execution of the instruction to load the accumulator and flags with the interrupt register status (LD A,I) or the interrupt refresh register (LD A,R).

This bug, which is apparently present in all masks of the Z80, was fixed in the HD64180 and Z180.

The precise consequence of this bug is that last issue's disable/enable interrupt routines will fail to re-enable interrupts on a Z-80 -- but only once in a (great?) while! This, of course, is one of the most elusive and difficult computer problems to track down -- it depends on the occurrence of a random event -- a real-time interrupt at precisely the moment this single instruction is being executed!

Having identified the exact problem, Don, author of two interesting TCJ articles on the iobyte and disk buffering, goes on to provide a robust solution. The key idea is that whenever an interrupt occurs, the cpu pushes the address of the next instruction (PC) onto the stack, so that the return from servicing the interrupt will resume the program code as if nothing has occurred. So, we first put a known value at that very spot in the stack. Then, we execute the "LD A,I" instruction and, before using the stack ourselves, test to see if the stack contents has changed. If it has, an interrupt must have occurred, and Don's code then forces bit 2 of the status flags to be set, thus correcting the wrong status.

Figure 4. Disable and Re-enable Interrupts

```
;
; Routine corrects Z-80 cpu bug and replaces version
; published in TCJ #38.  Routine assumed to be at or above 100h.
;
; Save interrupt status and disable interrupts
;
disable_int:
   push  af    ; save registers
   push  bc
   push  hl
   ld hl,-1    ; get SP address
   add   hl,sp
   ld (hl),0      ; put a nul there
   ld a,i      ; get interrupt status to F register
   ld a,(hl)      ; get byte at (sp) to A
            ; if an interrupt has just occurred,
            ; this will be the high byte of the
            ; address of this instruction
   push  af
   pop   bc    ; and into C
   and   a     ; was byte at (sp) changed?
   jr z,savit
   set   2,c      ; yes, correct z80 status bug
savit:   ld a,c           ; and save it
   ld (intflag),a
   pop   hl
   pop   bc
   pop   af
   di       ; disable non-maskable interrupts
   ret
;
; If interrupts were previously enabled,
; re-enable them.
;
enable_int:
   push  af    ; save register
   ld a,(intflag) ; if interrupts
   bit   2,a      ; .. were previously enabled
   jr z,1$
   ei       ; ..re-enable them
1$:   pop   af
   ret


intflag:ds  1
```

The code is shown in Figure 4. It assumes it is running at an address of 100h or higher, so that the high byte of the address is non-zero. Don further notes that the Z280's interrupt instructions are privileged and available only in supervisor mode. He therefore recommends testing for a Z280 (see this column, TCJ #37) and using a separate routine. This is first-rate analysis, and earns my enthusiastic nomination for coding-insight-of-the-month!

I also enjoyed talking with Don about his home-brew CP/M system -- it's built around a Z280 with 8-inch drives-- and I hope he will take up the challenge to write something about it for all of us. This chip, with a high-speed cache memory, paged memory manager, and privilege instruction set, holds exciting promise. But it has been held back by incompatibilities with some existing Z80 code.

Now, in the most recent masks of the new cpu, Zilog has eliminated earlier conflicts when running self-modifying code. This should make it possible to run ZCPR 3.4 and essentially all Z-System tools without modification.

(Continued on page 37)

C Pointers, Arrays and Structures Made Easier

Part 3: Structures in C

by Clem Pepper

Structures

When we truly comprehend the structure (and its kissing cousin, the UNION) we have planted our flag at the summit of the Mt. Everest of C. It is not that the structure is all that difficult, because it is not. Once we have acquired an understanding of the pointer and array, that is. It is a lack of know-how with respect to these that gives us those structure shudders. So the more we know of these fundamentals the more effective our programming with structures will become.

The index to the accepted "Bible" for the C language, *The C Programming Language* by Kernighan and Ritchie, lists fifteen (15)"structure" entries. While the distinction would appear to be tarnished by the 20 references to "pointer" and a tie of 15 with "array" this is not unexpected given that the definitive usage of pointers and arrays always precede those of structures. A genuine true-to-life structure is bound to include an array or two at the very least and heaven only knows how many pointer declarations will be spun out of it.

The authors dedicated one full chapter, the sixth, to the topic of structures. There are only eight chapters in the book, which should tell us something.

There might be the temptation, best resisted, to compare the array with the structure. The array, in an expansive frame of thought, is a structure of sorts. It is limited to a single variable type and is tightly defined in its structure. That is, we might say the type array practices a strict monogamy in contrast to the swinging lifestyle inherent to the type structure.

The structure is an aggregate type--a conglomerate of all types, including the structure. That is, an array may be declared as integer, character, float, double, or so on, but just the one or the other and no more. A structure template, on the other hand, may include declarations of any and all types from integer and character to array and embedded structure. With the structure we are able to write code which, when carried to the limit, may be incomprehensible even to its author.

A basic structure template, as it is called, looks somewhat like this:

```
struct tag {
      int     k;
      float   f;
   char arr[4];
           } ;
```

"tag" is an optional name for the structure. Any legal name can be used in place of tag.

The declarations inside the { }s following "tag" are referred to as structure "members." Observe that member types may differ--that is, our example includes an integer, a float, and an array. Wherein lies the structure's strength.

The ';' following the closing brace must be present. It follows an optional attribute. The example template does not include attributes. The attributes are variables assigned to the structure members. Since the members have little value without the attributes we know they will have to appear somewhere. Which they will. The meaning of attribute will be made clear in the example programs.

It follows that either the tag or the attribute is optional with the declaration, but not both. One or the other are required. For myself, I prefer to retain the tag even with attributes. In any event, if the declaration is global, and the attributes are defined within a function we really must have a tag to link the two together.

The rub with structures, in part at least, is that all manner of variants from this basic definition are possible. Right off, the closing brace and semi-colon are by no means the end of it. The unattended semi-colon, if you will recall, simply represents a NULL. It is replaceable with any number of statements, either singly or as blocks. Which is to say we can enter any number of attributes ahead of that closing NULL.

The declaration as shown may be local or global. If global, as frequently is the case, the attribute(s) may be either local (within a function) or global. As a starter, we can conceive of three potential schemes:

(1) global structure declaration with global attributes.

(2) global structure declaration with local attributes.

(3) local structure declaration with local attributes.

At this point we have a structure declaration, and declarations for individual members of the structure. But no more. That is, no memory has been allocated for any of these variables. Because there have been no assignments made. No assignment, no allocation.

A significant use of structures relates to lists. Lists typically involve a diversity of types. Names, addresses, and phone numbers are lists of import to us all. So suppose we look at an illustrative program based on these as shown in Listing 11.

For convenience in this first example the structure and all its associated quantities will be local, within main().

On compiling and running this program we see on our screen:

```
My friends are:
   Carlisle at phone no 003-6699.
   Jim at phone no 555-9990.
   Dottie at phone no 999-5602.
```

Some readers may cry "foul" in that this first example employs pointers. Well, we may as well get used to pointers, because they are going to pop up wherever we go in our experience with this most intriguing language.

The declaration is "static" because that is the only way to allocate memory to a structure declared within a function. Externally declared structures can be initialized without the static declaration. The declara-

tion is with reference to the variables, not to the tag.

The tag in this structure is "friends." It is not really required for this example as we also have the attribute "numbers[3]". We note there is no reference to "friends" anywhere in the program.

This structure has two members:

```
char *name;
char *ph_num;
```

The array declaration may appear a bit tricky. We are declaring three elements,

```
numbers[3] =  {
   ''Dottie'',  ''999-5602'',
   ''Jim'',     ''555-9990'',
   ''Carlisle'',''003-6699"
               } ;
```

but there are six strings in the assignment. How are we to explain this? Actually, there really are but three elements in the array. The trick is that there are two arrays. The first is:

```
numbers[3].name = { ''Dottie'',''Jim'',''Carlisle'' };
```

and the second:

```
numbers[3].ph_num = { ''999-5602'',''555-9990",''003-6699" };
```

The printf statement shows how this works in practice.

```
printf('' %s at phone no %s.\n'', \
   numbers[i].name,numbers[i].ph_num);
```

The dot is called the "structure operator." With the array we used an index number (or subscript) to identify a specific element. No such index exists for the structure. Here the dot relates a structure member to its attributes.

Note the plural, attributes. It is here where the programming power embedded in the structure becomes available to us. Once a member has been declared it can have as many attributes as needed.

While the dot will suffice for a great majority of our programs the time will come when a more potent operator is required. Rather quickly as a matter of fact if we are at all active in our programming.

The "pointer operator" for use with structures is the ->. Really resembles a pointer too, wouldn't you say?

Consider Listing 12 which is simply the next step in the evolution of this example.

This is the same program exactly except it now uses the structure pointer operator, ->, in place of the dot operator. However, there are significant differences in how the variable declarations are performed.

In our first example we made a point of not making use of the tag, friend. In this we must, as no attribute is included. The closing semicolon is required.

In place of the attribute we have declared a second structure

```
static struct friends numbers[3] =  {
   ''Dottie'',  ''999-5602'',
   ''Jim'',     ''555-9990'',
   ''Carlisle'',''003-6699"
                        } ;
```

The two declarations are related through the common tag, "friends." The members of this structure are the two arrays; there is no attribute. And lo--we have yet a third structure declared:

```
struct friends *pals;
   pals = &numbers[0];
```

Listing 11

```
/* STR_EX1.C ---------------------- **
** A first exercise in structures  */
#include <stdio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
/* == Begin program == */
main()
{
   int i = 3;
    static struct friends {
        char *name;
        char *ph_num;
                          }
        numbers[3] =  {
         "Dottie",  "999-5602",
         "Jim",     "555-9990",
         "Carlisle","003-6699"
                        } ;
      puts(CLRSCRN);
       printf("My friends are:\n");
      while(i--) {
printf(" %s at phone no %s.\n", \
numbers[i].name,numbers[i].ph_num);
                  }
         exit(0);
}
```

Listing 12

```
/* STR_EX2.C ---------------------------------------- **
** A variation of STR_EX1.C using structure pointers */
#include <stdio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
main()
{
int i = 3;
static struct friends   {
       char *name;
       char *ph_num;
                        } ;
static struct friends numbers[3] =  {
       "Dottie",  "999-5602",
       "Jim",     "555-9990",
       "Carlisle","003-6699"
                                   } ;
struct friends *pals;
  pals = &numbers[0];  /* pointer to array address */

    puts(CLRSCRN);
     printf("My friends are:\n");
      while(i--) {
printf(" %s at phone no %s.\n", pals -> name,\
pals -> ph_num); pals++;
                  }
         exit(0);
}
```

in which we declare a pointer, pals. We then assign this pointer to the beginning address of the array.

So now that we have this elaborate structuring of structures how do we make use of it? Through the common association with friends, that's how.

The assignment pointer, *pals, relates structure "friends" with structure "numbers." We will use "pals" to point directly at the array members we wish to access. To see this, compare the printf statements in the two listings.

```
Listing 11 -
   printf('' %s at phone no %s.\n'', \
   numbers[i].name,numbers[i].ph_num);
Listing 12 -
   printf('' %s at phone no %s.\n'',\
   pals -> name, pals -> ph_num); pals++);
```

We see there is no reference to "numbers[i]" in the second. In its place we find the pointer "pals."

In actuality, when we think of it, the while loop in both versions increments the array. The first directly with the index, the 2nd indirectly through the pointer.

Note that the declaration of *pals is tied to the structure "friends." Then the assignment pals = &numbers[0] links the structure declaration to the pointer.

The pointer/structure relationship is emphasized as it can be confusing. We must feel comfortable with the use of pointers if we are to make the best use of structures in our programs.

Now, for an exercise, relocate the declaration

```
static struct friends {
   char *name;
   char *ph_num;
                    } ;
```

external to main(). Is static necessary to the declaration or can it be dropped? Re-compile and run the program. What can we learn from this?

The next example, Listing 13, may come across as a bit on the heavy side. But we have to move right along, and with a bit of thought we should be able to interpret the action of this program.

The example intent is to exhibit a touch of the power available to us in the structure. In this program descriptions of two families are provided. They differ in a number of respects, including the total number of family members. The first family has five members, the second four. Two structures are declared in which the families are defined. Note that with these structures we could include any number of families but two will suffice for illustration. The second structure has another nested within it.

On running the program we will see this on our screen:

```
George has a wife, Sue, and 3 children.
George is 40; his wife Sue is 37.
Their children, Bill, Lynn, and Jean are 16, 13, and 9.
They get by on George's salary of $22000.

Phil has a wife, Marge, and 2 children.
Phil is 54; his wife Marge is 53.
Their children, Jill and Robert are 31 and 28.
They live well on Phil's salary of $45000.
```

The initial structure establishes the basic design of the family. Three parameters are defined: the family size (number of family members), the number of children, and the family income. Income is rounded off to the nearest dollar, so we use unsigned rather than float as the type.

```
struct family  {  /* first structure template  */
   int size;              /* First             */
   int no_chil;           /* structure         */
   unsigned int income;   /* members           */
             };
```

Now that we have defined the family in broad terms we need to get down to specifics. This is where the second structure enters in.

```
struct fam_des       {   /* second structure template */
   struct family det;  /* nested structure      */
      char dad[MAX_LEN];     /* Nested           */
      char mom[MAX_LEN];     /* structure        */
      char ch_1[MAX_LEN];    /* members          */
      char ch_2[MAX_LEN];
      char ch_3[MAX_LEN];
         int age[5]; };
```

The opening structure simply declares the design for the family. Note the absence of any reference to "family" in this. That reference, however, is prominent in the nested structure. In this declaration we are assigning names and ages. These are to be saved in arrays of ten characters or less in length. The maximum size of this family is five individuals. Note we could have made it 20 were that in our interest.

Because these are global declarations we need not include static in the declarations. That changes though when we make additional declarations inside main().

```
static struct fam_des home_1 = {
   /* initializing variable home_1 */
   5, 3, 22000, ''George'',''Sue'',''Bill'',''Lynn'',''Jean'',\
      40,37,16,13,9
                                  };
static struct fam_des home_2 = {
   /* initializing variable home_2 */
   4, 2, 45000, ''Phil'',''Marge'',''Jill'',''Robert'','''',\
      54,53,31,28
                                  };
```

Here we are declaring new structures related to the second external declaration, fam_des. We must provide a description for each family, not too surprisingly, of dad and mom and the kids and their ages.

But whoa! hold on there. The first three entries go back to the first declaration: family size, number of children, and the take home for their support. How's this for a clue?

```
   struct family det;  /* nested structure      */
```

We can't just ignore the first structure, and we don't. The declaration for the second is directly linked to it. So even though there is no reference to "family" in the local declarations, the link exists through the external declarations. And we thought pointers are confusing?

Now, suppose we increment the fun by replacing the dot operators with their pointer counterparts.

In this version (Listing 14) there are no changes to the external declarations. But as with Listing 12 there are significant differences inside main(). Instead of two local structure declarations we now find but one. An array yet.

```
static struct fam_des home[2] = { {
   5, 3, 22000,
''George'',''Sue'',''Bill'',''Lynn'',''Jean'',40,37,16,13,9
                                  }, {
   4, 2, 45000, ''Phil'',''Marge'',''Jill'',''Robert'','''',\
      54,53,31,28
                                   } };
struct fam_des *abode;
   abode = &home[0];
```

Again, suppose we relate the number of members in the nested structure "family det" ,(6), to the array assignments. Gosh, here home has only 2 array elements declared with 13 entries for the first and 11 for the second.

Well, we have got to back up all the way to the beginning to piece it all together. Let's see--we have family size, number of children, and income. No change there. Then we have dad, mom, and the kids. Again, familiar. Next comes everyone's ages which has its own array. So all is accounted for.

Again, we create a pointer, abode, related to the second structure, and assign its address to the zero element of the array home. Hopefully, this has a familiar ring to it.

The printf statements, as with Listing 12, utilize pointers. With abode pointing to home[0] printf displays the first family statistics. Incrementing abode sets it pointing to family number two. (We also insert an empty line for easier reading on our screen.)

```
printf(''%s has a wife, %s, and %d children.\n'',\
   abode->dad, abode->mom, abode->det.no_chil);
printf(''%s is %d; his wife %s is %d.\n'',abode->dad,\
   abode->age[0], abode->mom, abode->age[1]);
printf(''Their children, %s, %s, and %s are %d, %d,\
   and %d.\n'',abode->ch_1, abode->ch_2, abode->ch_3,\
   abode->age[2], abode->age[3], abode->age[4]);
printf(''They get by on %s's salary of $%d.\n''\
   ,abode->dad, abode->det.income);

abode++;
printf(''\n'');

printf(''%s has a wife, %s, and %d children.\n'',\
   abode->dad, abode->mom, abode->det.no_chil);
printf(''%s is %d; his wife %s is %d.\n'',abode->dad,\
   abode->age[0], abode->mom, abode->age[1]);
printf(''Their children, %s and %s are %d and %d.\n'',\
  abode->ch_1, abode->ch_2, abode->age[2], abode->age[3]);
printf(''They live well on %s's salary of $%u.\n'',\
  abode->dad, abode->det.income);
```

With some additional effort we could set up a new function just to display the family statistics. This would reduce the number of printf statements to a single set. Our experience with C combined with the information provided in this series should enable us to do this.

Our need now is to write some structure exercises of our own simply to firmly cement the concepts in our minds. As a starter, we might consider the three dominant US auto manufacturers: General Motors (GM), Ford, and Chrysler. Within GM we have Chevrolet, Pontiac, Buick, Oldsmobile, and Cadillac. Ford assembles the Ford, Mercury, and Lincoln. Chrysler the Plymouth, Dodge, and Chrysler. Each line has within itself a variety of models--Regal, Monte Carlo, Mustang, Bobcat, Fury, Dart, Continental to name a few. Locally there are dealerships for each. There is a listed price and a price to be bargained for. There is the basic car to which is added straight or automatic transmissions, stereos, air conditioning, power options, and so on. Man, there is more opportunity for structure exercising here than can ever be taken up.

Linked Structures

This is really a subject beyond the scope of this article. But because it is one we will frequently be coming in contact with a few words must be said on it.

Your basic text on C will most likely never mention the topic. C Primer Plus, one of the very best, devotes about half a page. However a really good treatment, with a "horsey" example, is given in "Advanced C Primer ++" (see reference list at articles end).

The basic approach is to create a series of structures. Each structure has two primary pointers. The first relates to the previous structure, the second to the current. The pointers are updated as new structures are added to the list.

Applications for linked structures are typically data files of an initially indeterminate length. These generally require organizing the data by some means, such as sorting, and the ability to add or delete entries.

The Union

The union falls into a category similar to the linked structure. Unions are set up in much the same fashion as structures. That is, there is a union template and a union variable. The dot and pointer operators are used as with the structure.

The unique ability of the union is the sharing of a common area of memory. A single area of memory is used to store a variety of data types. The size of the memory allocation is based on the requirement of the largest member of the union.

A union declaration is similar to that for a structure. As an example:

```
union two_mem {
   int mem_one;
   char mem_two;
            } mem_attr;
```

which looks very much like the structure declarations in the examples. The union lets us store different data types in the same memory space. To acquire a firm grasp on this we need a good understanding of how memory is allocated by our computer.
If our compiler is similar to Turbo C, Microsoft C or Power C we will find examples of the union in the dos.h library files. Where appropriate these express a typical declaration of

```
union REGS *inregs
union REGS *outregs
```

where the declarations are pointers to memory areas for holding entered and returned register values.

Structures Summary

We have learned that the structure is a type to be declared much as we declare the int or char or float. It bears some resemblance to the array in containing a multiplicity of data. But where the array requires all its elements to be of the same type the structure permits member variables to be of varying types, including other structures.

We saw the structure to be particularly valuable when dealing with lists. Declarations may be layered, beginning with the most general and working toward the most detailed. Declarations may be totally external, totally internal to a function, or divided.

We learned the use of two new operators unique to structures and unions: the dot and the structure pointer. We used these operators in programs in which variables declared in one structure were linked to variables in another, related, structure.

We were introduced to the advanced topics of the linked structure and the union. •

References for the Series

There are far too many texts and articles on the basics of C for a complete listing here. What follows are selections from my own library I have found of particular value to my needs.

Steve Schustack, *Variations In C*, Microsoft Press, 1985

Al Kelley and Ira Pohl, *C By Dissection,* The Benjamin/Cummings Publishing Co., Inc. 1987

Mitchell Waite, Stephen Prata, and Donald Martin, *C Primer Plus User Friendly Guide to the C Programming Language,* revised edition. Howard W. Sams & Company, 1987

Stephen Prata, *Advanced C Primer ++,* Howard W. Sams & Company, 1987

Stephen R. Davis, *Turbo C The Art of Advanced Program Design, Optimization, and Debugging,*" M & T Publishing, Inc. 1987

Power C--the High Performance C Compiler," Mix Software, Inc. 1988

Turbo C Reference Guide, Borland International, Inc. 1987

Listing 13

```
/* STR_EX3.C ------------------------------- **
** A multiple structure program illustration  */
#include <stdio.h>
#define MAX_LEN 10
char *CLRSCRN = "\033[2J"; /* ANSI screen clear */

struct family  {  /* first structure template  */
  int size;               /* First             */
  int no_chil;            /* structure         */
  unsigned int income;    /* members           */
              };
struct fam_des {   /* second structure template */
  struct family det;  /* nested structure      */
   char dad[MAX_LEN];     /* Second            */
   char mom[MAX_LEN];     /* structure         */
   char ch_1[MAX_LEN];    /* members           */
   char ch_2[MAX_LEN];
   char ch_3[MAX_LEN];
    int age[6]; };

/* == Begin program == */
main()
{
static struct fam_des home_1 = {
  /* initializing variable home_1 */
   5, 3, 22000, "George","Sue","Bill","Lynn","Jean",40,37,16,13,9
                              };
static struct fam_des home_2 = {
  /* initializing variable home_2 */
   4, 2, 45000, "Phil","Marge","Jill","Robert","",54,53,31,28
                              };
   puts(CLRSCRN);

printf("%s has a wife, %s, and %d children.\n",\
        home_1.dad, home_1.mom, home_1.det.no_chil);
printf("%s is %d; his wife %s is %d.\n",home_1.dad, home_1.age[0],\
        home_1.mom, home_1.age[1]);
printf("Their children, %s, %s, and %s are %d, %d, and %d.\n",\
     home_1.ch_1, home_1.ch_2, home_1.ch_3, home_1.age[2],\
     home_1.age[3], home_1.age[4]);
printf("They get by on %s's salary of $%d.\n",home_1.dad, home_1.det.income);

printf("\n");

printf("%s has a wife, %s, and %d children.\n",\
        home_2.dad, home_2.mom, home_2.det.no_chil);
printf("%s is %d; his wife %s is %d.\n",home_2.dad, home_2.age[0],\
        home_2.mom, home_2.age[1]);
printf("Their children, %s and %s are %d and %d.\n",\
     home_2.ch_1, home_2.ch_2, home_2.age[2],home_2.age[3]);
printf("They live well on %s's salary of $%u.\n",home_2.dad, home_2.det.income);

     exit(0);
}
```

Listing 14

```
/* STR_EX4.C ------------------------------------  **
** A variation of STR_EX3.C using structure pointers */
#include <stdio.h>
#define MAX_LEN 10
char *CLRSCRN = "\033[2J"; /* ANSI screen clear */

struct family  {  /* first structure template  */
  int size;
  int no_chil;
  unsigned int income;
              };
struct fam_des {   /* second structure template */
  struct family det;  /* nested structure       */
   char dad[MAX_LEN];
   char mom[MAX_LEN];
   char ch_1[MAX_LEN];
   char ch_2[MAX_LEN];
   char ch_3[MAX_LEN];
    int age[6]; };
/* == Begin program == */
main()
{
static struct fam_des home[2] = { {

    5, 3, 22000, "George","Sue","Bill","Lynn","Jean",40,37,16,13,9
                                  }, {
    4, 2, 45000, "Phil","Marge","Jill","Robert","",54,53,31,28
                                   } };
struct fam_des *abode;
  puts(CLRSCRN);
   abode = &home[0];

printf("%s has a wife, %s, and %d children.\n",\
        abode->dad, abode->mom, abode->det.no_chil);
printf("%s is %d; his wife %s is %d.\n",abode->dad, abode->age[0],\
        abode->mom, abode->age[1]);
printf("Their children, %s, %s, and %s are %d, %d, and %d.\n",\
     abode->ch_1, abode->ch_2, abode->ch_3, abode->age[2],\
     abode->age[3], abode->age[4]);
printf("They get by on %s's salary of $%d.\n",abode->dad, abode->det.income);

abode++;
printf("\n");

printf("%s has a wife, %s, and %d children.\n",\
        abode->dad, abode->mom, abode->det.no_chil);
printf("%s is %d; his wife %s is %d.\n",abode->dad, abode->age[0],\
        abode->mom, abode->age[1]);
printf("Their children, %s and %s are %d and %d.\n",\
     abode->ch_1, abode->ch_2, abode->age[2],abode->age[3]);
printf("They live well on %s's salary of $%u.\n",abode->dad, abode->det.income);

     exit(0);
}
```

Shells

by Rick Charnes

Hello, I hope by now you're all having a great time with your datestamping systems. I'd be very interested in hearing about some of the different and wonderfully creative applications people are finding. There certainly are a large number of utilities we in the ZCPR3 world have that take advantage of ZSDOS-style datestamping. The majority of these are not included on the ZSDOS distribution disk, so it comes to mind that those poor souls not involved with a Z-Node may be unaware of many of these. A tentative alphabetical list I made is shown in Figure 1.

I'd like to spend the bulk of this column delving into a general ARUNZ alias I wrote recently. As I've said time and time again, I consider ARUNZ to be the foundation of the ZCPR3 system, the ZCPR3 'language.' It's really not a special hot-shot alias but I hope it gives you a general taste of some special techniques that can be used and generally how this language works and will use it to create your own favorite aliases.

It is something I used when writing the twelve *.VAR files for last issue's column. I needed to determine the day of week of the first day of every month from 1989 through 1992. I didn't have any printed calendars near my desk to help me with this task. Then I remembered the wonderful ZSDOS utility ZCAL which will display a full calendar of any month. It suited my purposes perfectly. However, I quickly realized it was getting very tedious typing out for instance:

```
ZCAL 1 90;ZCAL 2 90;ZCAL 3 90;....ZCAL 12 92
```

one by one. There must be some way to automate this task!

OK, let's conceptualize for a minute. I'll try to reproduce my thought processes as they led to the alias' final form. I knew that ZCAL had to be run over and over again, but how could I arrange it so that the parameters fed to it could be regularly incremented so it would first display February 1990, then March 1990, then April 1990 and so on? What tool do we have for incrementing at the command level?

I had a feeling I knew how I needed to do it. There's a little-used, and I daresay little-known feature of REG.COM that can increment or decrement the values contained in the ZCPR3 registers. So we'll explore REG, and we'll also delve a bit into the wonders of ZCPR3's GO command. Figure 2 shows my alias STEPCAL (STEP through the CALendar).

STEPCAL will step through the months of a year one by one upon your pressing of the space bar. When it comes to December, it is smart and goes to January of the next year. Let's go through the alias.

First of all I should say that I arbitrarily chose to have the syntax be simply 'STEPCAL<cr>' rather than 'STEPCAL <parm1> <parm2>'. Usually it's fairly easy to code an alias to allow for either form: parameters or no parameters. However, here we're using ARUNZ's 'user input' feature which complicates matters slightly.

Let's see how the prompted input feature works. When Jay Sage modified and enhanced the way this works with ARUNZ version 09L it opened up tremendous possibilities. Any text between the $" and " symbols is echoed to the screen as a prompt. This happens before any commands of the alias are run. You can have up to four prompts in the form:

```
$'' This will be prompt #1"$'' This will be prompt #2"$''
Prompt #3"
```

etc. Don't be deceived by the "$" you see in the above line. The first " is the termination of the previous prompt and the $" is the beginning of the next prompt; no intervening space is necessary though I could have put one in for clarity. The responses the user makes to these prompts are stored in buffers which you can use later in the alias. You can reference a single token (anything separated by a space), a single command, an entire command line, etc. Here we will use the symbol that represents a token since that is all the user will (presumably!) enter. We have two prompts in the alias. The symbol to represent the user's response to the first prompt is $'e1 and that to the second prompt with $'e2. So with that in mind, let's see what we have.

The first thing that happens is the prompt:

```
YEAR to begin (last two digits only)
(or <CR> to resume:)
```

appears on the screen. Notice the way we get a part of a prompt to appear on a new line is by putting '^m^j' in our prompt text. For this example we'll say we want to start at February 1990 so we enter '90.' Any response such as '89' or '92' or '74' would be appropriate as well. Then we hit <CR> and:

```
MONTH to begin (or <CR>):
```

appears as the second prompt. For our purposes we enter '2'. (We must use the ordinal number rather than the name of the month.) We hit <CR> and the actual commands start. Before I explain what happens, let me first note that the

```
STEPCAL $"YEAR to begin (last two digits only)^m^j(or <CR> to      <<
        resume:)  "$"^m^jMONTH to begin (or <CR>):  "if ~nu $'e1;  <<
        and ~nu $'e2;reg s1 $'e1 q;reg s0 $'e2 q;zif;              <<
        echo f%>irst, this month's display...;zcal;/step2

STEP2    $zfi;go $rt00 $rt01;if in ^[)[<SP> %>or %<<CR> %>to       <<
        advance, 'n' to quit] ^[(;if reg 0 < 12;reg8000 p0 q;      <<
        else;reg8000 s0 1 q;reg8000 p1 q;fi; $0;fi
```

Figure 2: Alias to step through the calendar.

APPEND	Concatenate files like PIP, but inserts the date of concatenation as a line of text between each file.
ARUNZ	Allows for display, manipulation and other use of several date-related parameters in macro command scripts; $dd is the current date, $dm the month, $dy the year $dh the hour, $dn the minute, etc. Extremely useful and flexible. For me the most powerful of them all.
COPY	For many, the standard copy utility. Preserves all date information. Very sophisticated date handling; if destination file already exists, COPY compares datestamps and asks for further instructions accordingly.
CRUNCH23D	With the datestamping version of CRUNCH, we can embed the date stamps in the compressed file itself (it's not dependent on the directory entry) and we can thus transfer datestamp information via modem.
DATSWEEP	Like NSWEEP with datestamping features added. Displays create, access and modify date of all files. Can copy or archive files by date using '=', '<' or '>' operators ("copy all files modified in the last two weeks of January"). Can manually change a file's datestamp. Very feature-laden program.
DBASE	Yep, we have a patch for this venerable program. No longer will you have to enter the date when the program loads as you've been doing for years. ZSDOS will do it for you.
DIFF	Compares files by date or otherwise. Very sophisticated.
FILEDATE	Perhaps the most exciting of them all. A directory utility that not only displays the creation, access and modification dates of files in a directory, but is sensitive to and can sort by date. Options to sort from most recent files to oldest, oldest to newest, list only files modified or created today, only modified/created since a given date, etc. It truly opens up a whole new world of computing.
LBREXT	Preserves a file's datestamp (create, access and modify) upon extraction from a LBR). LOG Perfect way to keep track of how long you spend on each "type" of task on your computer. Will "log" time spent on each of several self-chosen or given categories into a file for later viewing and displaying. Excellent for record-keeping for IRS.
LPUT	Preserves a file's datestamp upon insertion into a LBR.
LSH	Our new standard command history shell by Rob Friefeld. Very nice to see its display of the time at its command prompt.
MCAT, XCAT	As of early April still in beta-testing but may be out by the time you read this. Complete floppy or hard disk cataloging system; ZSDOS version of Irv Hoff's classics. Allows user to display to the screen, for example, all files in one's collection of floppy disks created before July 1988, or during the month December, 1987, etc. When used normally, will display files' datestamps along with their names.
MCOPY	Copy program allowing simultaneous copy of multiple unrelated files using filelist feature of ZCPR3.
MEXPLUS	Scripts can display, wait for and measure time. I consider my greatest programming feat to be the MEXPLUS script I have mentioned previously in this column to log on to PCP Pursuit. ZCPR3's shell variable system then keeps a non-volatile time log of monthly use. The script can tell you how long you have been on a BBS, how long you were on the last logged BBS, the current time, how long you have been attempting to connect to PCP, how long was the last attempt, etc.
NT	Loads and saves virtually instantly, Rob Friefeld's NT is a quick "note"-maker and mini-editor. It begins all files with the current date and time at the top.
PPIP	Copy program, sensitive to archive bit if desired; always preserves date of source to destination.
PRINT	Prints date and time at top of each printed page.
SDD	Similar to FILEDATE, but more limited. Only displays filedates, cannot sort by date.
TD	Displays current date and time on screen.
UNCRUNCH	Uncompresses the file and restores its datestamp.
WAIT	Will "wait" until a specific time before performing a given operation. Useful for ZEX scripts and other multiple-command line operations.
ZCAL	With no parameters, displays a calendar of the current month with the current day highlighted. With parameters it will display a calendar of any month in the 20th century.
ZDE	For me the granddaddy of them all. The only wordprocessor that supports ZSDOS. ZDE is the ZCPR3/ZSDOS version of VDE. Automatically preserves a file's creation date after editing.
ZFILER	Extraordinary file management utility shell. Like NSWEEP taken to the 100th power. Displays current time on the screen, preserves datestamp of copied or "moved" files.
ZMANAGER	Another file utility menu/shell, displays current time on screen, etc.
ZSVSTAMP	Absolutely essential when using editors and word-processing programs other than ZDE. Without this program, users of all other editors will find that after working on a file its 'creation date' has suddenly become today's date, even if they started working on the file months ago! ZSVSTAMP stores this stamp information temporarily in a buffer at load-up, then re-imprints it onto file upon exit. ZXD Another directory program. Good for general use but really shines for those using DOSDISK or P2DOS type datestamping.

Figure 1: List of ZCPR3 utilities.

eagle-eyed command buffer limit watchers among you may be wondering, "Isn't he living rather dangerously with that impolitely long first prompt? With our 200-character limit, he should be conserving space for the actual command part of the alias! As a matter of fact, looking at this alias with my editor I see it stretches out well over 210 characters already! How can he possibly expect it to run?"

Ah, have no fear, O sharp-eyed one. For there is a crucial aspect of ARUNZ's prompted-input feature that bears considering: the prompt itself is **not** placed into the command buffer at all! Why should it -- it's not a command! It is simply text that ARUNZ writes directly to the screen. As far as I can gather the command processor is not involved in the process at all. It is similar to ZEX's '^< text ^>' feature -- it is direct console output. So do not worry about long prompts -- those who have seen my MAKLIB12 alias (available at finer Z-Nodes everywhere) know what kinds of tremendously sophisticated and attractive and L-O-N-G things you can do with prompts.

I believe there is no theoretical limit to the allowed length of a prompt string, so have all the fun you want with video codes, cursor positioning, whatever -- to be creative is the most important thing. I run into a practical limit, however, that has nothing to do with ARUNZ. Turns out that VDE, which is what I use to do all my alias work, can not display text past column 255. I know, I know -- you're thinking someone would have to be crazy to write anything that long, but if you're not a little bit crazy you don't belong using something like ZCPR3. Believe me, I have run into this limit on several occasions when doing really fancy stuff with user prompts. I either compromise or -- go to Word-

Star.

In any case, prompted text is not put into the command buffer and does not count towards our magic 200-character figure beyond which we must not venture.

Back to the alias. Next we check to make sure anything was entered into response to the prompts; this will become important later as we will see. 'IF ~NU $'E1' means 'if the response to prompt number 1 was not a null response' -- in other words, if something was indeed entered. We check the same for prompt number 2, and then we come to the part where we set registers -- always a favorite activity of mine. As I have said before, one of the beautiful things of ZCPR3 is its ability to carry messages from program to program, and those 16 little bytes starting at 30 bytes after the start of the message buffer otherwise known as registers are instrumental in that task. Here we use the REG command to set register 1 ('S1') to whatever was entered into response to the first prompt. The trailing 'q' indicates 'quiet mode' and means that nothing will be displayed to the screen. Since we entered '90' register 1 will be set to that value. Note that any value up to '255' is possible.

Since we entered '2' in response to the second prompt, the next command sets register 0 to the value '2'. 'ZIF' terminates the IF testing and restores the flow state to 0.

Now we come to one of the unique parts of the alias. In writing this initially I was reloading ZCAL from disk each time. Then I remembered about our wonderful GO command, which reloads whatever is at 100h in memory, the place where programs normally execute. But then I had a problem: if I was going to use the GO command to run it most of the time, how could I get it to load in the first place? I solved this by making two aliases, the first as a 'dummy load' to place ZCAL at 100h in memory in the first place, and the second to run it the majority of times with GO.

So I chose to display the current month's calendar ('ZCAL<cr>' with no parameters) with a ECHO status message to that effect, and then move on to STEP2, the second alias.

First we have a '$z' at the beginning of the second alias which tells ARUNZ to flush any commands left in the command buffer when the alias runs. I like to do this just in case there is any danger of overflowing the buffer with too many characters (always remember that we have a limit of 200). When we see later how we repeat STEP2 over and over again, it will become urgent to flush the buffer each time or we will quickly get an OVFL message. Next we flush the IF level -- which we will be setting later -- with 'FI'. Now we come to our command 'GO $rt00 $rt01'. Remember that GO will rerun whatever program is residing at 100h, so we have one important consideration here. Since we are running a new alias, didn't we just run ARUNZ? In that case, wouldn't that get rerun instead of ZCAL?

Yes, it would -- if ARUNZ was indeed run at 100h. But this was the whole purpose of the development of type-3 and type-4 programs. We can now run programs at locations in memory other than 100h. So in order for this alias to work you must make sure to have either the type-3 version of ARUNZ which runs at 8000h or the type-4 which runs at an address determined at run-time by the configuration of your own system.

The symbols '$rt00' and '$rt01' translate into the values held in registers 0 and 1 respectively. Note that Jay Sage, ARUNZ' author, has given us a great deal of flexibility in determining how these figures will display. The letter 't' after the 'r' indicates that we want them to display as 'T)wo' decimal digits. We also could have chosen 'o' for O)ne, 'h' for H)ex or 'f' for F)loating decimal. Remembering the values to which we set registers 0 and 1 in the previous alias, our command resolves to

```
GO 02 90
```

As we saw, GO will re-load ZCAL and hence our command will be:

```
ZCAL 02 90
```

which of course displays the calendar for February 1990.
OK, now we come to the part where we want to increment the first ZCAL parameter. First, however, we must consider that perhaps at this point the user wants to abort the alias. A good way to do this is with the IF INPUT command.

```
IF IN <SP> OR <CR> TO ADVANCE, 'N' TO QUIT
```

works perfectly. A touch of the SPACE key or <CR> are both equivalent to entering 'Y' and will set the IF level to true and continue the alias, while entering 'N' will set the IF level to false and skip all the commands until ELSE or FI, which in our case effectively terminates the alias.

Now, let's assume we have entered <SP> or <CR> and set the IF level to true. If register 0 is less than 12 (at this point it is '2') we then increment register 0. In our case it would set to '3'. The 'p0' parameter to REG.COM does that, or we can alternately use the syntax '+0'. Notice, however, that I am using the type-3 version of REG.COM running in memory at 8000h which I have renamed REG8000.COM. For reasons that will become apparent shortly, we cannot use the version that runs at 100h. The resident REG contained within an RCP would, however, be fine. Again, the IF level terminates and the '$0' symbol resolves into the name of our alias, 'STEP2,' and the alias runs again.

Incidentally, it's a good idea to use the symbol '$0' in your aliases when you want them to repeat, rather than the actual name. There are several reasons for this. You might later change the name of the alias and forget that you have included this command to re-invoke it as part of the command sequence. If you had previously used its actual name you would soon find yourself looking at your error handler and scratching your head. Secondly, when you use ARUNZ's 'multiple-name' feature the '$0' symbol is indispensable as it can cover for whatever particular name you enter from the command line. For instance, we could have entered:

```
STEP,CAL=CAL,STEP $''YEAR to begin....
```

as the beginning of our alias. We could then have entered 'STEP', 'STEPCAL', 'CAL', 'CALSTEP' or any of several other names at the command prompt for the name of our alias and **all** would have been properly re-executed by our '$0' symbol. Lastly, using '$0' saves a couple of bytes of disk space in our ALIAS.CMD file and we're slowly learning how important it is for all of us to be ecological.

Looking at the beginning of the alias and the GO command, we can now understand why it was necessary to use either the RCP version of REG or the version that loaded at 8000h. With the GO command we now want ZCAL to run again. If, however, we had used a 100h version of REG, it would be REG that would re-execute with the GO command, since GO executes whatever resides in memory at 100h. Everything involved in this alias except for ZCAL should be either memory-resident running at 8000h such as IF.COM, REG.COM and ARUNZ itself. If so, when we come back to the beginning of the alias and the GO command, ZCAL at 100h will still be undisturbed and it will be that program that is executed by GO.

So now our first command with GO runs the command line

```
ZCAL 03 90
```

and we are displayed the calendar for March 1990. We are again presented with the "<SP> or <CR> to advance, 'n' to quit" message. We will hit the SPACE bar, register 0 will be incremented again, and the

alias will again re-execute. Each time we hit <CR> or <SP> register 0 will increment and we will be presented with the next month's calendar, until....register 0 is up to 12, meaning it's December. In this case we want ZCAL to display January of 1991. No problem. In that case, when we come to the command in the alias, 'IF REG 0 < 12' the flow state will be FALSE and we will skip to the commands immediately following the 'ELSE'. Here, REG.COM will set register 0 back to 1, **and** something special will happen. Register 1, presently containing the value of '90', will now be incremented to 91. When the alias repeats, the GO command at the beginning will now be running:

```
ZCAL 01 91
```

and we can go from January through December once again, always relying on register 0 for the ordinal value of the month and register 1 for the year.

When we want to exit from this 'loop,' we simply hit 'n' at the prompt and the IF level is returned to 0 and we exit the alias. A nice thing of this technique is that if later we want to repeat the calendar-stepping, our system 'remembers' where we were when we left off, since we are relying on the registers to store information. In that case, we respond with two <CR>'s at the first two prompts. Then the IF level after the two first 'IF ~NU...' commands will be FALSE and we will not set the registers to any particular value but rather leave them where they are.

I really enjoy using the registers in this fashion. I also like this technique of doing one 'dummy' load of the crucial program in question, ZCAL, in order to perform later rapid memory-based re-invocations using the GO command. It's particularly fascinating to note that you can perform several other commands later in the alias but still have the GO re-execute ZCAL as long as all of the other commands are either memory resident or type-3 transients running at 8000h.

Debugging Aliases

Two tips for debugging aliases: (1) Put 'MU3 $+E0100' as the first command in your alias when you're testing it. (Make sure it's the type-3 version of MU312.COM running at 8000h.) When your alias runs, MU3 will be the first program to load. Its parameter '$+e0100' translates into 100h past the beginning of your environment descriptor and it will be this address that MU3 will display as its first screen. The ZCPR3 command buffer begins at 0104 bytes past the environment descriptor, so starting at the fourth byte past the beginning of MU3's display you will see a visual representation of your alias is it is about to run. You may count 200 bytes past this point and that will be your limit. In this way you'll be able to 'see' all resolutions of all ARUNZ symbols before they actually run; this has been extremely helpful to me when working on complex aliases.

(2) When you want to print them out to get a look at them ordinary word processors are generally no help. Quite often, your lines inside ALIAS.CMD are 150+ characters in length. Solution: use the classic ZCPR3 utility PRINT.COM. There is a feature that is not even documented in our erstwhile bible Richard Conn's *ZCPR3: The Manual* that is very helpful for printing out files containing long lines of text. Whenever PRINT.COM finds a line greater than the TCAP-defined printer width it will print as much of the line that fits up to this point. It will then skip a single space and insert the two characters '<<' immediately to the right of the line. These characters are inserted by PRINT.COM indicating that more of the line is to follow. It then wraps to the next physical line on the printer page and continues printing the rest of our line. It will 'break' your line into as many physical lines as necessary. Lines of over 200 characters in length are easily printed in this fashion and are clear and readable, being broken up on the page by PRINT.COM.

For additional clarity and especially when printing out more than one alias, I also recommend using PRINT.COM's 'L' option ('PRINT ALIAS.CMD L'). This will number your aliases and indent all 'broken-up' lines just slightly in from the first, providing a wonderful degree of visual ease and a great deal of help in writing and debugging long multi-alias sequences.

I like writing aliases.

Conclusions

Thought of the month: why don't word processors come with an internal command to move the cursor to the last letter of the current word? It seems I want to do this almost as much as I go the first letter in the next word.

Some advertisements for recent contributions to our ZCPR3 public domain are in order. We have a lot of nice work finally being done with the straight-line graphics about which I've been ranting recently. Hal Bower, who has been doing a yeoman's job in our community for some time now, has put together a new extended TCAP that takes advantage of this capability of many terminals. If your terminal has graphics character capability, pay your local Z-Node a visit and pick up his TCAP-HB4.LBR. He's also rewritten the venerable and classic GRDEMO to use his new TCAP, which for years was the Z-System world's most sophisticated demonstration of graphics capability. GRDEMO is now being challenged by ... Eugene Nolan of the Philadelphia area who has done a wonderful job putting together a library and demo program of some very sophisticated graphics windowing routines that also rely on Hal's new TCAP. Look for Gene's SWINDO15.LBR on the Z-Nodes ... Edward Barry took pity on me (bless his heart) with my complex *.VAR files I described two columns ago and wrote DW.COM. DayofWeek uses ZSDOS calls to calculate the day of week and sets ZCPR3 register 0 to 1 through 7 for seven days of the week. This greatly simplifies one's *.VAR files. Pick up Edward's DW.LBR at quality Z-Nodes everywhere ... Greg Miner of Nova Scotia has filled a real need. His LOCNDO (LOCate 'N' DO -- cute) is something that's been crying to be written. How many times have you run FF.COM to look for a file, then when it's found you curse at having to type all the extra keystrokes to actually do something with the program you've just found? I mean computers are supposed to make life easier for you. For instance, suppose you're not sure on which directory you left that article you started for The Computer Journal. You type 'FF TCJ.ART' and FF.COM finds it. Then (yawn) you have to enter 'VDE TCJ.ART' again to actually look at it. What a waste. Now you need only enter:

```
LOCNDO TCJ.ART VDE $
```

LOCNDO will search your disk until it finds TCJ.ART. It will then substitute 'TCJ.ART' for the '$' in its command line and run VDE on it -- all in a single command. Very nice and very simple. Thanks, Greg.

That'll about do it for this bi-month. Be sure to write with comments and suggestions. Z you next time. •

Real Computing

The National Semiconductor NS32032

by Richard Rodman

Once upon a time, there was a man who lived near a lake wherein dwelled a fearsome crocodile. The man liked to fish in the lake, but was afraid of the crocodile. So, he made a deal with the crocodile. Before fishing, he would bring a delicious croissant and throw it into the lake. The crocodile would eat it, swim to the other side of the lake and leave the man alone.

For years, the man enjoyed fishing peacefully at the lake. But one day, the crocodile didn't come up to retrieve the croissant. The man wondered, but began fishing anyway. Suddenly, the crocodile bit him in the seat of his pants. "Why", he cried, "your croissant is in the lake, as usual!"

"Yes," admitted the crocodile, "but I'm cutting down on sweets."

The moral of this story is that the underlying architecture always comes up to bite you. If you're programming some kind of segmented, memory-modeled processor, you can try to hide from it with C all you like, but sooner or later, you're going to find its teeth firmly embedded in your posterior.

The 32532 microprocessor

The 32532 is the current top-of-the-line of the NS32 family. It's completely software-compatible with the other members of the family, but has a number of improvements.

First, the MMU has been moved on-chip. The MMU, as far as address translation is concerned, is identical to the 32382 MMU. It uses 4 kilobyte pages and can address up to 4 gigabytes. The breakpoint logic has been changed again, however. The control bits have been taken out of the Master Control Register and put into a special Debug Control Register.

Second, the 32532 has a four-stage internal instruction pipeline which allows for overlapped execution of instructions. In particular, operands for the next one or two instructions can be fetched while the current instruction is processed. The pipeline includes logic for doing "branch prediction." Since the future state of the machine for conditional branches isn't known, backward conditional branches are predicted as occurring, and forward ones as not occurring. This matches statistical measurements on actual code. Another pipeline-related issue is memory-mapped I/O. To solve this problem, the 16-megabyte region from FF000000 to FFFFFFFF has been dedicated for memory-mapped I/O, and the pipeline logic will insure that the reads and writes will take place in the order coded.

Third, there are on-chip caches: a 512-byte instruction cache, and a 1024-byte data cache. This is in addition to the instruction pipeline. An I/O decode signal is provided so that I/O references are not cached. Also, a new instruction, CINV (cache invalidate) is provided so that either single entries or entire caches can be invalidated, for example, when the page tables are changed during a context switch.

Fourth, five new bits have been defined in the CFG register, mainly for controlling the operation of the memory caching.

The 32532 incorporates all the features of the 32332, such as the dynamic bus sizing and burst-mode memory access. The chip is 1.6 inches square and sits in a 175-pin PGA socket. It is available in 20-, 25- and 30-MHz versions. It runs on a single clock--the TCU chip is not required.

Simply put, the 32532 is the current apex of CISC MOS microprocessor design. How fast is it? I hesitate to cite benchmarks, because they have been hopelessly compromised by the RISC hoopla, but National says the chip can compute 10 real VAX (CISC) MIPS.

How can you get your hands on this CPU? There are actually a number of new options. First, National has a two-board VME board set that'll set you back about $10,000. Heurikon has been advertising their VME board under the banner "Ten MIPS and No RISC" for around $4,000. There are two PC coprocessor boards out, one from Opus and one from Aeon Technologies, but they're not for hobbyists, either. Lastly, there's the 32532 Designer's Kit.

The 32532 Designer's Kit

I found out in late January that there was a special offer on the 32532 Designer's Kit for $532, due to end January 31. It wasn't a time for thinking, it was a time for action.

So, I became the proud possessor of the NSV-532DK. What comes with this kit? The following:

- A NS32532U-25, 25MHz processor chip, and a 175-pin, machined-pin pin-grid-array socket for it
- A NS32202N-10, 10MHz ICU (interrupt controller chip).
- Two PROMs containing three ROM monitors: TDS, MON-32 and a RAMless monitor.
- A PAL chip and a digital delay device.
- A bare, 4-layer, silk-screened PC board (11" x 11-1/4")
- Several books, including manuals on the monitors, the *Series 32000 Programmer's Reference Manual* by Colin Hunter, data sheets on the 32532 and the 32381, schematics and instructions.
- A diskette containing one file, TDSCOM.COM, a very crude downloader program for the TDS monitor.

Note that there are NOT enough parts here to build the computer. You need about 256K of CY7166 16Kx4 bipolar RAM chips, plus about $150-$200 worth

of other sockets, chips, capacitors, SIP resistors, and connectors. The board can be populated with a Weitek math chip and 32580 or with a 32381 floating-point unit. You also get a coupon for a discount on a 32381.

Populating the board will result in a simple computer with 256K bytes of high-speed RAM, two serial ports, and some parallel ports.

Programmer's Reference Manuals

I have been remiss in not taking time to discuss data books. Let's compare three leading databooks on popular microprocessors:

- Intel *80386 Programmer's Reference Manual*, Anonymous. About 390 pages. Pages on instructions: 174. Examples: 7. No index.
- Motorola *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, Anonymous. 218 pages. Pages on instructions: 107. Examples: None. No index.
- National *Series 32000 Programmer's Reference Manual*, by Colin Hunter. About 340 pages. Pages on instructions: 217. Brief examples of each instruction and addressing mode. 6-page index.

By any objective measure, the National book is clearly superior. But then, we wouldn't judge a processor by its data book, would we? Or, perhaps, a company's concern for its customers?

Just as a test, give me, in 5 minutes or less, an example of an 80386 BTR instruction, complete with binary encoding.

The 32CG16 microprocessor

The 32CG16 is a stripped-down version of the 32016 with high-performance graphics commands added into the instruction set. It is intended as a dedicated graphics controller or coprocessor, such as for a laser printer or video-display board.

What instructions have been added? BITBLT (bit-block transfer), with AND, OR, and XOR options; EXTBLT, an "external BITBLT", which uses an external BITBLT processing unit (BPU); BITWT, bit word transfer, for purposes such as text generation; MOVMPi, a pattern-fill instruction; and bit-string instructions, TBITS and SBITS.

Contrary to what you may have heard, it is not necessary to use National's graphics chip set with the 32CG16. The only instruction which makes use of it is EXTBLT.

Because the 32CG16 is intended as a dedicated device, it doesn't have logic for supporting the MMU chip. It can, however, be used with the 32081 floating-point chip. It has an on-chip clock oscillator (so, again, the TCU is not needed) and comes in a 68-pin PLCC package.

The 32CG16 is compelling because it gives you the high performance of a specialized graphics controller such as the 34010, yet is lower in cost, much easier to program, and has lots of development tools (any NS32 compilers or assemblers can be used).

I understand that the chip was developed in association with Canon Corp. as part of a laser printer controller chip set. However, by taking this part, an ICU, 4 video RAMs and a video DAC, it should be pretty easy to construct a nice video graphics coprocessor. The ICU would be set up to interrupt the processor at horizontal and vertical sync times. At these times, the processor would sequence logic to generate the correct sync and burst signals, and issue the "load row" strobe to the video RAMs.

Next time

I'm going to keep my options open for the next column. Maybe I'll have a status report on the 532 system, or on the free OS, or some new hardware that's coming out, or some great deals on older hardware. Until then, watch out for crocodiles!

Where to write or call

Aeon Technologies
90 S Wadsworth Boulevard
Lakewood CO 80226

Heurikon Corp.
3201 Latham Drive
Madison WI 53713

National Semiconductor Corp.
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara CA 95052-8090

Opus Systems Inc.
20863 Stevens Creek Boulevard
Building 400
Cupertino CA 95014

Richard Rodman
8329 Ivy Glen Court
Manassas VA 22110
BBS: 703-330-9049

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, DosDisk; Plu*Perfect Systems; Clipper, Nantucket; Nantucket, Inc. dBase, dBase II, dBase III, dBase III Plus; Ashton-Tate, Inc. MBASIC, MS-DOS; Microsoft. WordStar; MicroPro International Corp. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C; Borland International. HD64180; Hitachi America, Ltd. SB180 Micromint, Inc.

Where these, and other, terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

News

Interactive Text

I have been working with a very interesting product from Patch Panel Software called "Figment--the Imagination Processor." It is interactive story development software which provides a complete environment for developing, debugging, and publishing interactive fiction.

Figment provides window-managed play-back display, object-oriented style of development, rule-based logic programming, natural language command structure, and context sensitive editing process. There is a game map and a player map, and revision 2.30 adds user definable attributes, user defineable link types with values, character profiles, and character viewpoints, and overlays.

The story logic represents a knowledge base defining appropriate behavior for different situations so Figment can be called a "Situation Modeler." It can be used by lawyers, for example, to create a courtroom simulator; by salespersons to produce a role-playing sales trainer; by writers to model character and plot development; and by educators as a creativity tool to teach logic development. Stories developed with Figment might be simple stories with multiple endings depending on player choices or they can be full role-playing fiction games.

Writing interactive games or stories is a great way to relax after a hard day of programming--it's much better than just playing someone else's game. There is also a market for stories, games, or role playing training programs. I recommend Figment for either entertainment or for developing salable products.

Figment runs on IBM-PC and compatibles with DOS 2.01 or later, and 384K of RAM (640K advisable). No special display adapter is required. Figment is not copy protected, and is available for $95 from Patch Panel Software, 11590 Seminole Blvd., Seminole, FL 34648, phone (800) 543-0731.

dBXL Diamond Release 1.2d

My favorite DBMS has been revised to include even more options. One of the most significant changes to me is that they have improved the SET RELATION command so that it is now possible to SET more than one RELATION from the same work area. This will be very useful for an application on which I am now working.

Some of the other new features are: SET COMPATIBLE determines whether the maximum number of fields in a database is 128 or 512 (the previous version was limited to 128 fields); SET SWAPPING specifies whether your dBXL work session is swapped out of memory when you use the DOS, RUN, and ! commands; SET KEY specifies a program or procedure dBXL executes when you press a specified key; REPLACE MEMO replaces the text of a specified Memo field with the contents of a specified Array variable and STORE MEMO stores the text of a specified Memo field into a specified Array variable; ALIAS() returns the alias of a specified work area, ELAPSED() returns the number of seconds elapsed between two specified time expressions; FCOUNT() returns the number of fields in a database; INDEXKEY() returns the key expression of an index; INDEXNAME() returns the filename of an index; LENNUM() returns the display length of a Numeric expression; MEMOLINE() returns a line of text from a specified Memo field; MLCOUNT() returns the number of lines of text in a specified Memo field; READVAL() returns the value of the current @...GET memory variable or field; and UPDATED() indicates whether and @...GET memory variables or fields were edited during the last READ, EDIT, CHANGE, APPEND, INSERT, or BROWSE. This is just a partial list, there are more.

These enhancements, combined with the features which dBXL already had, make it a very nice development tool--it's the one I grab when I need a database for vendors, subscribers, etc. It supports most dBASE code, but adds the enhancements which provide much more power. There is a dBXL/LAN version, and a companion Quicksilver compiler which is also available in a UNIX version.

dBXL is available for $249 from WordTech Systems, Inc., 21 Altarinda Road, Orinda, CA 94563, phone (415) 254-0288.

Database Reference Books

One of the reasons why I choose dBASE-like tools such as dBXL is that there is a great wealth of reference material. I need this material because I am still learning the intricacies of data and information management. I don't need books which just repeat the command and function definitions found in the manuals, I need guidance on what it it that the system should do. As stated in *Expert dBASE III Plus* "Most published discussions of dBASE programming begin and end with how to put commands together in a command file. That's important, of course, but it's a little like teaching a house builder how to hammer nails into a board without teaching what board to nail, or where, or why."

I find that I am relying on three books for most of my reference, and was surprised when I noticed that they are all published by the same publisher, SYBEX.

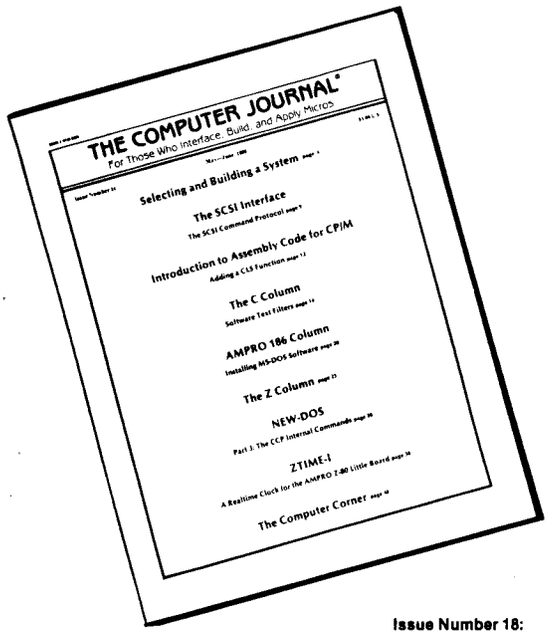
The three books are:

Expert dBASE III Plus, by Judd Robbins and Ken Braly. This book provides a lot of detail on what it is that a database should do, and then shows how to do it with dBASE.

Mastering dBASE III Plus--A Structured Approach by Carl Townsend. This book uses an accounting database for its example, and includes much information on design, and implementation.

dBASE III Plus Programmer's Reference Guide by Alan Simpson. This is a large book, over 1,000 pages, and is the one I use to get the full details of the dBASE language. The commands and functions are grouped by type instead of alphabetically, and there is supporting text to fully explain the operations and options.

I recommend these books, and since my top three choices are from SYBEX, I'll certainly check them first for other needed books. •



THE COMPUTER JOURNAL

Back Issues

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 6:

- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System. Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS--Anatomy of an Operating System, Part 2.

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

#39

THE COMPUTER JOURNAL
190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

(Continued from Page 22)

Keep the Mail Coming!

We TCJ columnists thrive on your suggestions and objections. The inevitable production lag of a journal means that when you first receive TCJ there is barely time left to submit copy for the next issue.

Do you have another slant on the issues covered here? What topics would you like us to take up next time? Sit down at the keyboard and write, or pick up the phone -- now, while your ideas are churning.

A Trap Door Puzzle

Assuming the following code is in memory, what's left in memory after the the code at 100h has executed?

```
org     38h
pop     de
push    hl
rst     38h

org     100h
ld      sp,(0006)
ld      hl,0C7C7h
rst     38h
```

How would you load it into memory? Test it? •

(Continued from page 2)

users. Computers are simply following the marketplace in this regard." With articles such as "The Meaning of Good Source Code," "What Makes Tailoring Code Difficult," "Vertical & Diagonal Markets & the UNIX System," "How to Capitalize on Vertical Markets," and "Ten Rules for Selecting a Vertical or Diagonal Market," it is very helpful reading--even for people who are not now using UNIX.

More applications are going to have to operate as toolkits which can provide different features at different times as the needs change.

A wordprocessor is a good example. When I am working on a large novel of a thousand pages or more, I want an editor which is very fast so that I can work without losing my train of thought. I don't need formatting, because I have no idea of what page design the publisher will select. I don't want it to natter at me if I misspell a word, I'll take care of that later. I just want to get words and thoughts on disk. Partial sentences. Jumbled up syntax. Just get the thoughts down for later nit-picking. For hard copy print out I just want word wrapped text with paragraph breaks--no centering, no justification, no underlining, no boldfacing, and no condensed or expanded width characters. None of the fancy features which all of the vendors are rushing to incorporate.

For producing the magazine, the requirements are entirely different. Then I want to see line and page breaks with the type font and page design I'll use for the final output. I'll also want to see titles, subtitles, headers, footers, and pagination in actual size and position. I also need an output program to drive the laser printer to produce exactly what I've seen on the screen.

These are two different needs, but I don't want to learn two different sets of commands. I want one program which can be tailored to my needs. I may work on the novel for two hours, then on the magazine for an hour, then write a few letters, and return to the novel. I'll also squeeze in some C programming. What I want is something which can change as I change my tasks. I don't do only one type of work such as a secretary in an office might do, so the program has to be able to handle a wide variety of jobs. But, I am uncomfortable with a large, slow, clumsy program which tries to do everything for everybody at the same time.

I would like to see a system that contains separate modules for editing, page preparation, and runoff. The modules would be able to be customized for the current task, and used separately or together as needed. My ideal system does not exist, and probably never will exist. As the wordprocessors add more page preparation features, and the page preparation programs add more editing features, the differences between them will diminish and systems which can do the entire job will evolve.

Users' toolkits are also very much needed in other areas, such as data and management information systems. There are database development programs such as dBXL, FoxBASE, Clipper, and Paradox, and there are canned accounting programs such as general ledger and accounts receivable. But, there are many application specific requirements, such as order entry and processing, or customer contacts, which can not be met with canned programs. Small businesses cannot afford (or won't pay for) professional programming assistance. They need a toolkit which will allow them to create their own custom program without getting involved with programming details.

There are application generators which claim that they will generate the database application you need. We will be taking a very close look at them in order to determine just how extensive of a system they can generate. Everyone has their wordprocessor and spreadsheet, and DTP is becoming mature. Data and information management is the next application which will be opened up to the user. The demand is very real and very widespread--it will be a very fruitful field for those who write the software and templates for the user.

Portability

The consumer demand for portability is increasing. Not only the ability to transfer data which I discussed in issue #38, but also the transfer of skills, and programs. Software of the future will not be written for any specific operating system. You will be able to slap in the disk (or what ever other storage media we use then) with your favorite wordprocessor, and run it, regardless of who made the hardware, what CPU it uses, or the details of the operating system. The software will make general calls, and all hardware systems will include the code to convert those calls to conform to their operating system specific requirements. Something like what the P-System tried to do, but this time much more efficient and faster.

There will still be system specific software which can utilize non-standard system characteristics for critical dedicated applications, but the majority of end user software will run on any system. The only people who will be concerned with the details of the individual systems will be the programmers who design and code the system specific special applications.

I'm not sure that this will be entirely a good thing, because it will force mediocrity and stifle development.

Standardization is necessary for mass markets. Our cars all have the steering wheel on the same side (at least here in the U.S.), and the brakes are a pedal on the floor instead of some being a lever on the dash and others a pull cord from the roof. A reasonably experienced driver can drive any standard commercial car, but not an Indy 500 racer.

The computer industry is going to have to standarize whether we like it or not. •

(Continued from page 40)

68HC11

I mentioned before how my boss had acquired a Motorola 68HC11 evaluation unit. In the literature was a flier from New Micro Forth. Well I spent $40 and bought a ROM for the unit. This is not a full evaluation but a contest board. Motorola has contests from time to time and gives these boards away. Their objective is to sell chips, and giving whole systems away to get people to use their products works perfectly for them. I have been playing with the board and have had lots of fun.

After I got my Forth ROM and manuals from New Micro, I took it to one of our local Forth meetings. I had a LED display attached and a variable resistor feeding the A/D converter. The objective was to write a Forth routine to convert the A/D signal to a LED output. The only information I gave them was the output and input port address. Three people tried it and all had something working in 15 to 20 minutes. The point of interest was how each person took a different approach to the problem. We all had a great time watching each other and plan on bringing in a project every month from now on.

The New Micros Forth works great. I needed to control a stepper motor for some tests and so used their unit to program it. My stepper was working in a hour and I had a complete program in two nights of playing around. I have included the screens just to show how little it takes to make it work in Forth. I found some C source for the same operation and was appalled at how much coding was required. For our testing the ability to change variables on the fly was actually more important than anything else.

I like the New Micros products and software. I found the manuals to be excellent despite a few minor typos. For a beginner the explanation on starting Forth is excellent. Their way of handling word definitions takes a few minutes to understand, but once mastered worked well for me. I find it close to F83 so users of that Forth will need little book work. There are a few bugs I found, and the issue 9 of "More on NC4000" has a bug list in it (by Bill Muench, pp. 22, printed by Offete Enterprises, (415)574-8350 or from FIG, $15).

If you are looking for embedded system controllers where you may need to change parameters on the fly, New Micros has several products for you. One of our members has his $99 computer controlling a modem at home. This allows him to call home and test software ideas he is working on at work. They use Forth to gather and sort data from a nuclear power plant. This is done remotely over the phone lines, and the little 68HC11 running Forth emulates the data they want to get from the power plant.

```
Forth code for stepper motor operation on  New Micros 68HC11:

HEX  ( this means accept numbers in  HEX format )
FF B004 C!      ( these steps are needed to shift the )
C004 1C !       ( dictionary operation from the 68HC11 )
50 1E !         ( internal RAM memory to the external )
C060 22 !       ( larger 8K memory space.   )
FORGET TASK     ( Now that you have more memory space )
C080 DP !       ( we reset the line length, and dictionary )
00 B030 C!      ( pointers, followed by making all the I/O )
FF B007 C!      ( lines in known states, in our case turn the )
FF B003 C!      ( motor phase drivers off. )

VARIABLE DELAY ( this starts the stepper program by first )
01 DELAY C!     ( defining variables and filling them with values)
VARIABLE CURPOS ( there is more than one way to do this, I am )
0 CURPOS !      ( storing values into the the variable )
VARIABLE PHASE ( alternately you can place values into the next )
0 PHASE C!      ( dictionary location by using CREATE and comma - the )
                   VARIABLE DIR?  ( choice is yours to make )
0 DIR? C!       ( DELAY is how fast the stepper pulses occur )
VARIABLE #STEP ( CURPOS is current position in pulses )
0 #STEP !       ( PHASE which stepper motor phases to power )

CREATE PTBL     ( make a table of what values to send )
EF7F ,          ( to actual I/O port for each of the phases )
DF C, BF C,     ( of the stepper motor - 4 )( using comma here )
                ( and using C, 16 and 8 bit stores into dictionary )
: DELAY1 ( - ) DELAY C@ 0 DO LOOP ; ( this is our delay time )
: DOSTEP ( - ) PHASE C@ PTBL + C@ B003 C! ; ( this steps the motor )
: PCHK+ ( - PHASE ) PHASE C@ 1 + DUP 4 = IF DROP 0 THEN ; ( go positive )
: PCHK- ( - PHASE ) PHASE C@ 1 - DUP -1 = IF DROP 3 THEN ; ( go negative )
          ( phase is 0 to 3 so must roll over to 0 from a 4 )
: GETPHASE ( - ) DIR? C@ 0= IF PCHK+ PHASE C!
                            ELSE PCHK- PHASE C! THEN ;
     ( getphase determines direction then up/down  phase table pointer )
: MOVE  ( - )  #STEP @ 0 DO GETPHASE DOSTEP DELAY1 LOOP ;
     ( move the motor x number of steps using do loop )
: CHKPOS ( NEWPOS - DIR )  DUP CURPOS @  DUP ROT <
                            IF - #STEP ! 0
                               ELSE SWAP - #STEP ! 4 THEN ;
( check position value and adjust to provide number of steps
needed  by subtracting current position from desired position )
: CUR+ ( - ) #STEP @ CURPOS @ + CURPOS ! ; ( update current position )
: CUR- ( - ) CURPOS @ #STEP @ - CURPOS ! ; ( update in negative direction )
: SWDIR  ( DIR - ) DUP DIR? C! 0= IF CUR+
                                    ELSE CUR- THEN ; ( set direction flag )
: GOPOS ( NEWPOS - ) DUP CURPOS @ = IF EXIT
                                    THEN CHKPOS SWDIR MOVE ;
( update the  position variables, check for proper position,
change direction, and  then move the stepper to new position,
exit if no change in position )
: NEW ( n - ) GOPOS ." CUR " CURPOS @ U. ." STEP " #STEP @ U. ;
( this is the main loop of the program - you enter 45 NEW and the
stepper will move 45 steps in  positive direction. If 10 NEW is
entered next, it will go negative or back 35 steps. NEW will
print on the screen the current postion value and the number of
steps taken. Delay is adjusted to prevent loss of step pulses -
if too fast motor doesn't move. Start DELAY at a large value -
then scale back for desired speed. I use lots of variables
instead of stack operation due to need to change the values -
more stack usage would be faster - but harder to change and to
follow for the beginner. This also makes it a little more like
Object Oriented programming. The ( - ) means nothing on the stack
is used by the program. )
```

More CAD

I am still using ORCAD for schematic work. Lately I have been doing the printed circuit designs and using ORCADPCB. The PCB program I am afraid to say is not as good as the schematic cap-

(Continued on page 39)

(Continued from page 38)

ture part. I have found lots of bugs and some hard to deal with problems. They have promised an updated version next month. One problem that has cost me a lot of time is the text operation. This is to put text on the silk screen. The first time I found the problem was trying to change a device part number, I could not do it. The next time I wanted to add text, like a copyright name, and found the steps to be almost impossible to master.

I find ORCAD easy to use for almost everything and it has been a great program for most operations. My solutions for the text problem however involves using DEBUG and changing the HEX file data. This has been the only way I have been able to change text once the board design has been created. My best word of caution is not to layout the board without being very close to the final design. I laid out at least six or seven boards before I got down which steps to do in which order. For a product costing $1500, I feel the problems don't reflect the cost. For that much money you would expect better than you seem to get. If they continue to provide free updates, good telephone support, and you can work around the problems, then the cost may be justified.

Last time I said they didn't allow for printer output in PCB. They gave me the name of a company that has a program that converts plotter data to Epson printer output (actually most common printers supported). We got the program and it works great. It is faster than any graphic layout program I have seen. It is called FPLOT and worth every part of the $64 they charge. We have used it with ACAD and feel it might work with desk top publishing as well. Actually it will work with any program that can produce HP-GL plot files, including your own. Give them a try.

Parting Shots

As I draw this column to a close, I would like to comment on the rising tide of UNIX backing. By backing I mean media articles that keep saying how Unix will be your next operating system. I know UNIX has plenty of advantages, and as system memory and hard disk capacities increase it will find more homes. But to say that most users will be using it is ridiculous. The established base of DOS users will not change, and as yet UNIX has not shown anything that can provide users the type of interface for so few dollars as DOS.

Enjoy computing and try some of the little micros, they make great fun projects. I would like to hear from people about their little projects. If we get enough interest, we may even have a contest for the most interesting and worthless design. The project's only redeeming value should be educational. Something that does bell ringing and light flashing based on the color of a shirt. WHY? only for FUN! •

Products Mentioned

FPLOT Corp.
Suite 605
24-16 Steinway St.
Astoria, N.Y. 11103
(212) 418-8469

TURBO PASCAL INNOVATIONS

Hot off the Press!

A book for anyone programming in Turbo Pascal who wants to learn more. Improve your software projects by discovering the tricks of user interface design with Rick Gessner; consider the question of code as data with well-known writer Jeff Duntemann in his chapter on procedural types; learn to refine the timing of the code you produce and to precisely predict run time from Chuck Burchard; follow Joe Sroczynski through a development of DOS management techniques; learn a **lot** about 5.0 graphics from Dr. John Figueras; work with David Kutzler on the fascinating history of calendar theory and follow his algorithm development for a very thorough dates unit; and finally, learn from Lane Ferris' superbly written unit which allows you to develop TSR programs that multitask with ordinary DOS programs.

. . . presenting the best that several intelligent persons have to offer. . . diverse and useful.
Dr. Dobbs Journal (Sept 1989)

More

All of that and more, because you also receive all of the source code on an MS DOS disk which comes with the book at no extra charge. Included on the disk are some additional FREE programs that are yours to use: a README file called TYPERITE.EXE, a de-archiving program (UNPAK.EXE), and a DOS shell called DISKTOOL.EXE that allows you to COPY, MOVE, DELETE, RENAME, and VIEW files.

$32.95
Postpaid in USA, add $2.50 for surface shipment outside the USA.
Visa and MCard accepted

ORDER FROM:
The Computer Journal
190 Sullivan Crossroad
Columbia Falls, MT
59912
(406) 257-9119

$32.95 Includes MS DOS Disk

The Computer Corner

by Bill Kibler

Fatherhood--I hope all those who have become recent fathers are getting more sleep than I am. As you can guess my son came and my sleep vanished. I am still working but find it hard at times. This really hasn't slowed my work down as we get closer to the software stage.

Product Stages

It occurred to me the other day, how many stages a product goes through before becoming real. We started out talking with the prospective company and determining their needs. From there you make an educated guess at what products you will use, along with their cost. You roll your dice and guess at how many hours it will take to do the design, product selection, programming, hardware layout, and proto-typing. To this time table you put a cost, added 20% for profit, and then add 20% for errors and over runs.

Looking back at my boss's bids and what has been happening, I can see that product selection is more important than any other stage. Your choice of devices can make or break the product. I have spent 90% of product selection on three items and all in twice our estimated time. The real killer was a vacuum sensor. The system requires a narrow range of vacuum readings and most sensors are for 1 PSI up. We needed 67 to 150 MTORRs. If you are like me, the terms PSI has some meaning, but MTORRS?

I now know the difference between PASCALS, TORRS, PSI, MBAR, and Inches of mercury. I didn't know we had so many ways of measuring air pressure and vacuums. It turned out our needs fell between cheap devices for measuring air pressure and the expensive vacuum measuring systems. I found some items costing more than our whole product. The answer was to use a standard sensor with our own circuit. The sensors we needed were not very expensive, nor big sellers, but companies making the units were not willing to give me much information.

After spending several weeks going around with the manufactures, I finally got a support tech to tell me which pins needed what. That was the sum total of information available on the sensor I chose. No fancy literature packets, no reams of charts, and no samples of circuit usage. What I did was buy a complete meter unit and dissect it. That combined with proto-typing a circuit answered the design questions. This stage took almost as much time as we had allotted for the entire product selection process.

The CPU

Not all product or design selections are problems, the CPU was simple. I considered several types and needed one that had a A/D converter, EPROM, enough I/O lines and be cheap. The choice is Motorola's 68705R3. The support I can get from Motorola is many times better than those selling vacuum devices. I found they also have a BBS on line to get free software for their CPUs. [(512)891-3733 (8N1) 300-1200-2400].

I looked at many other devices, but cost or design limits prevent their use. Motorola is the leader in small controllers, in my mind that is. The reason being their support and price structure. I looked at some Intel versions and their cost was prohibitive and they lacked any form of support. Support is Motorola's strong point.

I have talked with several Motorola technical support people and have gotten quick and straight answers. A similar discussion with Intel resulted in more questions than answers. Motorola's support for their MCUs (as they call them--Micro-Controller Units) is a line of proto-typing boards. These boards are about $500 each and can emulate an embedded MCU. They usually have a monitor program to control emulation and RAM to use in place of the MCU's ROM.

Their next support item is frequent training packets. We just got the one for their 68HC705 series and if we complete all the course work it may cost us nothing. The sale price is $87 over their regular $165. My only problem is we are using the 68705R3 and NOT 68HC705s. If you can't see much difference between those two device numbers, you are a bit like me. After the box came, I realized there was a difference between the devices.

Motorola has two lines of products going for them. Their old line consists of HMOS devices and the new line is HCMOS based products. All of these are based on the original 6800 internal design. The process of construction and pin layout are the only differences. So you need to pay attention or you might miss the "HC" signifier. The "HC" means it is HCMOS based and the pin out will be different. The clock speed is half as well, but the chip functions similarly other than that. The only programming difference is a multiply instruction which came with the 6801 devices.

The 68HC705 series is a relatively new line for Motorola and the purpose of the training package is to get more people interested in their products. I found the internal programming information the same and helpful for any of their 6805 based devices. So even if you are not interested in the 68HC705, their training course might be very useful. The board that comes with the unit makes a great programming station as well as prototype system. Motorola's bigger evaluation units are needed however if you really want to get into run time emulation operation without programming a device.

(Continued on page 38)